Games on Dynamic Networks

Routing and Connectivity

Frank G. Radmacher

Games on Dynamic Networks

Routing and Connectivity

MV WISSENSCHAFT

Frank G. Radmacher, »Games on Dynamic Networks: Routing and Connectivity«

Typeset by the author in Arno Pro using LaTeX.
Figures and cover picture by the author.
Cover design by MV-Verlag.
Printed and bound by MV-Verlag.

ISBN 978-3-86991-775-7

D 82 (Diss. RWTH Aachen University, 2012)

Games on Dynamic Networks: Routing and Connectivity

Von der Fakultät für Mathematik, Informatik und Naturwissenschaften
der RWTH Aachen University zur Erlangung des akademischen Grades
eines Doktors der Naturwissenschaften genehmigte Dissertation

vorgelegt von

Diplom-Informatiker
Frank Georg Radmacher

aus Düsseldorf

Berichter: Universitätsprofessor Dr. Dr. h. c. Wolfgang Thomas
Juniorprofessor Dr.-Ing. James Gross
Universitätsprofessor Dr. Berthold Vöcking

Tag der mündlichen Prüfung: 24. Februar 2012

Zusammenfassung

Unendliche Spiele sind ein mächtiges Modell für die Analyse dynamische Netzwerke, die fortwährend topologischen Veränderungen ausgesetzt sind. In diesem Rahmen vertreten die Spieler die sich entgegenwirkenden Kräfte im Netzwerk. Diese Dissertation behandelt drei verschiedene Zweipersonenspiele, welche sich insbesondere auf das Garantieren von Konnektivitäts- und Routing-Eigenschaften richten. In jedem Modell muss ein Spieler die Funktionstüchtigkeit des Netzwerkes herstellen, während der Gegner Ausfälle und Lasten erzeugt, die während des Betriebs auftreten.

Im ersten Teil betrachten wir *Sabotage-Spiele*, die van Benthem 2002 einführte. In diesen Spielen durchquert ein *Runner* einen Graphen und versucht einen Zielknoten zu erreichen, während ein *Blocker* Kanten entfernt. Wir verfeinern dieses Spiel auf zwei Arten: Zum einen betrachten wir eine allgemeinere Gewinnbedingung, die in linearer temporaler Logik angegeben wird. Zum anderen untersuchen wir die Variante, in der Blocker durch einen probabilistischen Spieler ersetzt wird. Wir zeigen, dass in beiden Fällen das Entscheidungsproblem, ob Runner gewinnt, PSPACE-vollständig bleibt.

Im zweiten Teil entwickeln wir ein *Routing-Spiel*, in dem ein *Routing-Spieler* Pakete an ihre Zielknoten ausliefern muss, während ein *Lasten-Spieler* ununterbrochen Pakete erzeugt und Verbindungen im Netzwerk für bestimmte Zeit blockiert. Wir beweisen grundlegende Grenzen für das Berechnen von Routing-Strategien; wir zeigen jedoch auch algorithmische Lösungen auf. Die Ergebnisse hängen sowohl von der gewünschten Routing-Eigenschaft als auch von der Gröbe des Modells ab. Für bestimmte Szenarien entwickeln wir realisierbare Routing-Algorithmen, welche jeweils eine Routing-Eigenschaft gegen jedes mögliche Verhalten des Lasten-Spielers sicherstellen.

Im dritten Teil führen wir ein *Konnektivitätsspiel* ein, in dem ein *Constructor* gegen einen *Destructor* antritt. Während Destructor Knoten löscht, kann Constructor Knoten wiederherstellen und unter bestimmten Bedingungen sogar neue Knoten erzeugen. Auch modellieren wir den Informationsfluss im Netzwerk, indem wir Constructor erlauben, die Beschriftungen benachbarter Knoten zu ändern. Als Ziel hat Constructor entweder eine Erreichbarkeits- oder eine Sicherheitsbedingung, d. h., Constructor muss

entweder ein zusammenhängendes Netzwerk herstellen oder aber sicherstellen, dass das Netzwerk immer zusammenhängend bleibt. Wir zeigen, unter welchen Bedingungen das Lösen dieser Spiele entscheidbar ist und untersuchen in diesem Fall die Berechnungskomplexität. Die Ergebnisse hängen von den Fähigkeiten von Constructor ab und unterscheiden sich für das Erreichbarkeits- und das Sicherheitsspiel.

Abstract

Infinite games are a strong model for analyzing dynamic networks that encounter continuous topological changes during operation. In this framework, the players represent the contrary forces which modify the network. In particular, this thesis deals with three different two-player games which focus on guaranteeing routing and connectivity properties in dynamic networks. In each model, one player has to establish the proper operation of the network, while the adversary produces failures and demands that occur during operation.

In the first part, we study *sabotage games*, which van Benthem introduced in 2002. In these games, a *Runner* traverses a graph and tries to reach a set of goal vertices, while a *Blocker* removes edges. We refine this game in two ways; namely we consider a more general winning objective expressed in linear temporal logic, and we study the variant in which Blocker is replaced by a probabilistic player. We show that in both cases the problem to decide whether Runner wins remains PSPACE-complete.

In the second part, we develop a routing game in which a *routing agent* has to deliver packets to their destinations, while a *demand agent* continuously generates packets and blocks connections for a certain amount of time. We show general limitations for obtaining routing strategies but also point to algorithmic solutions. The results depend on both the desired routing property and the coarseness of the model. For certain scenarios we develop feasible routing algorithms, each of which guarantees a routing property against any behavior of the demand agent.

In the third part, we introduce a connectivity game between a *Constructor* and a *Destructor*. While Destructor deletes nodes, Constructor can restore or even create new nodes under certain conditions. Also, we model information flow through the network by allowing Constructor to change labels of adjacent nodes. Constructor either has a reachability or a safety objective, i.e., Constructor has to either establish a connected network or guarantee that the network always stays connected. We show under what conditions the solvability of these games is decidable and, in this case, analyze the computational complexity. The results depend on the abilities of Constructor and differ for the reachability and the safety version.

Contents

List of Figures

List of Algorithms

Introduction

In the past two decades, one of the most significant changes in technology as well as in our daily life was driven by the rapid development and pervasive use of computer networks, as for instance mobile phone networks and the Internet. Nowadays, even stationary computers can often only accomplish their tasks when an intact network connection is available. In computer science, this implies a shift in the view of a computer system. It does not suffice anymore to see a system as a fixed entity that only reacts to possible inputs of an environment; rather the entire network must be seen as a dynamic system which performs tasks, while the network itself changes continuously.

The dependency on the reliability of a dynamic – and therefore rather fragile – network strongly calls for formal methods to guarantee its proper operation. However, it is not hard to believe that the shift from static systems to dynamic networks makes the formal analysis and verification much harder, and it is not even clear under what conditions one can guarantee certain network properties at all. In this thesis, we address problems on dynamic networks, and we focus on analyzing a wide range of routing and connectivity properties. To reflect the dynamic nature of networks, we choose a game-theoretic model, and we consider dynamic behavior that results from adversarial agents. Generally speaking, we model dynamic networks as two-player games of infinite duration. One player usually has to establish the proper operation of the network, while an adversary generates failures and demands that occur during the operation.

In the following, we motivate our game-theoretic approach in more detail and present the models of dynamic network games with which we deal in this thesis.

Dynamic Networks via Games

The idea to model a reactive system as an infinite two-player game dates back to the late 1950s. Church (1957, 1963) considered a scenario where a system gets an infinite input sequence from an environment letter by letter. The system produces an infinite output sequence by responding to each input letter with an output letter. The task for the system is to ensure – against any choices of the environment – that the sequence of inputs and outputs fulfills a given specification. Church raised the question of whether a finite-state controller for this task can be constructed algorithmically for a given specification – a problem to which is nowadays often referred to as *Church's synthesis problem* (for instance, see Thomas, 2008b, 2009).

This problem indeed has the format of an infinite two-player game, more precisely the form of a *Gale-Stewart Game* (Gale and Stewart, 1953). One player represents the environment; he generates an input letter with each of his moves. After each move of the environment, the system reacts with an output letter in her move, and so on.[1] The sequence generated in this way forms an infinite play of this game. Each play that fulfills a given *winning condition* is won by the system; the environment wins all other plays. If a player has a strategy to win every play – against any choices of the adversary – this player has a *winning strategy*. Maybe contrary to intuition, for very exotic winning conditions it may not be the case that one of the players has a winning strategy. However, it is a classical result from the theory of infinite games, that this is the case for a wide range of winning conditions; especially, this holds if the set of winning plays belongs to the Borel hierarchy (Martin, 1975).

Translated into the framework of infinite games, the task for Church's synthesis problem is, first, to decide whether the system has a strategy to win, and second, if the system wins, to algorithmically compute such a winning strategy. The first solutions to this problem were found independently by Büchi and Landweber (1969) and Rabin (1969) for ω-regular winning conditions. Many seminal results on solutions for various specifications and system models followed, especially with applications to the verification and synthesis of reactive systems. For an overview (and also

1 Throughout this thesis we follow the convention that one player is female and the other one is male. We always associate the female player with the *constructive* player, who tries to satisfy a given specification.

for an introduction to infinite games) we refer to the tutorial by Thomas (2008a) and the book by Grädel, Thomas, and Wilke (2002). A first case study of automatic hardware synthesis was done by Bloem et al. (2007).

A dynamic network is very similar to a reactive system in the sense that it has to response to demands and that it has to compensate changes to the network structure. The demands have usually the form of generated network packets. The network has to cope with the generated packets by routing them to their destinations. Changes in the network structure may be caused by link failures, e.g., due to interference, or by node failures, e.g., due to client or server breakdowns. The network has to response on failed links and nodes by forwarding packets via alternative routes or by establishing new links and nodes in some self-healing process.

The view of a dynamic network as a usual reactive system has one shortcoming. A reactive system is a single entity which produces an output sequence in the same way as the environment provides an input sequence. In contrast, in a game on a dynamic network, only the environment can be seen as an entity but not the network itself. Modeling the environment as a single, malicious player reflects the worst-case scenario, in which all events that disrupt the proper network operation can happen everywhere in the network at the same time. The nodes of a dynamic network, however, are rather autonomous; and each of the nodes itself can be affected by failures and demands. Furthermore, the nodes can only response to environmental changes in a local, but possibly coordinated, way. Therefore, we need a new notion of infinite games which embraces the asymmetric behavior of the players as they occur in dynamic networks.

In this thesis, we study three different games for modeling dynamic networks. The first two games deal with routing properties, whereas the third game deals with connectivity properties. In both cases the network is considered as a graph. The vertices are the network nodes, and an edge between two vertices indicates a direct link between these nodes. Edges can be either directed or undirected, depending on the desired model. Usually, we assume that edges are undirected. If not stated otherwise, our results hold for networks with both directed and undirected edges.

The first game is due to van Benthem (2005); we extend his model and provide several new results. The two remaining games are our contribution to model more realistic scenarios of dynamic networks. In the following, we describe the three games in more detail.

Runner vs. Blocker – Sabotage Games

The first model describes a simple scenario in which only a single packet is routed through a network in an adversarial environment. Although many theoretical models for this scenario exist, we focus on the *sabotage game*, which was first proposed by van Benthem in 2002 and published in 2005. We obtain the sabotage game in various ways as a special case of the more complex games that we study in this thesis. So, the results on sabotage games help us to tackle some of the obstacles which we later encounter in our studies of more complex game models.

The two players in a sabotage game are called *Runner* and *Blocker*. The game is played on a graph, which possibly contains multi-edges. Runner traverses the graph (starting from a designated *initial vertex*), while after each of her moves, Blocker deletes a single edge from the graph. In the very basic version of the sabotage game, Runner wins if she reaches some vertex in a given set of final vertices. In contrast to the more complex game models, a play of a sabotage game is always finite, because the players cannot move anymore after all the edges have been deleted.

Routing Agent vs. Demand Agent – Dynamic Network Routing Games

With *dynamic network routing games*, we address two shortcomings of sabotage games. First, instead of permanent edge removals, blocked edges can become available again; and second, instead of considering the routing of a single packet, packets which have to be routed simultaneously are generated again and again. The game is played by a *demand agent* and a *routing agent*, who act in alternation. The demand agent claims the demands of both the environment and the network clients. This means that he blocks edges in the network, which become available after a certain number of turns (if not blocked again in the meantime); he also generates packets in the network, each of which has a fixed source and destination node. To prevent the demand agent from blocking all edges and generating an unmanageable number of packets in every turn, the actions of the demand agent are subject to given constraints. These constraints may be subject to both the channels that are currently available and the packets that are currently in the network. The routing agent transmits each packet via a non-blocked edge in every turn, but she is not allowed to use a single edge

to transmit two packets from one node to another node in one turn.

The plays of a routing game are infinite in general. We consider several winning objectives for the routing agent: the delivery of every packet to its destination, the delivery of each packet within a given number of turns, the boundedness of the number of undelivered packets in the network, and the combined form where each packet has to be delivered while the number of undelivered packets in the network has to stay bounded.

A possible application of these routing games is to realize dynamic spectrum access in mobile networks. The basic idea is that the clients in a network (also called the *secondary users*) use frequency bands that are locally not used by their owners (the *primary users*) for a certain period of time. However, when a primary user requires one of his frequencies, this frequency becomes unavailable for the secondary users in the transition range. In our model, an unavailable frequency corresponds to some blocked edges in the secondary network. Dynamic network routing games allow to study whether, and under what conditions, one can provide certain guarantees ("quality-of-service") for routing in the secondary network. More precisely, by determining the winner of a routing game, we can check whether a routing objective is achievable under certain demands (which are both the frequency usage in the primary network and the packet load in the secondary network). Moreover, a winning strategy for the routing agent describes a routing scheme for the secondary network clients to react on changing demands instantly.

Finally, it should be mentioned that the routing game model can be used for similar applications apart from computer networks. We may think of a traffic network where connections may be blocked due to road works or accidents, or a network of distribution centers of a packet delivery service whose delivery vehicles may fail.

Constructor vs. Destructor – Dynamic Network Connectivity Games

Instead of guaranteeing routing properties, in many applications it may be sufficient to ensure that the network nodes can establish a connected network or to guarantee that the network nodes always stay connected. Our third model is a framework for studying such connectivity properties of dynamic networks, which have self-healing capabilities and may also allow the extension of the network beyond its original shape. A *dynamic*

network connectivity game is played by two players called *Destructor* and *Constructor*, who move in alternation. The faults that can occur in the network are represented by Destructor; he tries to disconnect the network by disabling a network node in every turn. Constructor can be seen as the player who represents *users* and *supplies* of the network. The users exchange information via the network. The suppliers maintain the network topology; they can restore deactivated nodes and even create new nodes. Since the task of both the users and the suppliers is to provide a connected network, we combine the capabilities of both parties into the player Constructor. In contrast to Destructor's node deletions, Constructor's actions are local and subject to a given set of rules. On the one hand, to restore or create nodes Constructor needs special maintenance resources, called *strong nodes*. On the other hand, the labels of the nodes that are involved in a local action of Constructor must match some rule of the given rule set.

The set of rules may consist of three different kind of rules; each rule describes a possible action of Constructor. A *relabeling rule* allows Constructor to relabel two adjacent nodes. This rule models the information flow through the network evoked by the users; nodes and edges stay fixed. With a *movement rule* Constructor can shift a strong node to an adjacent node, if the labels of the two nodes match the rule. The strong node can also be shifted to a deactivated node, which is then restored. Finally, a *creation rule* enables Constructor to use a certain number of strong nodes to create a new node, if the labels of the strong nodes match this rule; the new node is then connected to the strong nodes which created it.

A play of a connectivity game is an infinite sequence of networks that arises by the actions of Constructor and Destructor. As winning objective for Constructor we consider the connectivity of the network 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.

Besides the described scenario of a dynamic network that should be maintained by users and suppliers, connectivity games can also be seen as an approach to verification and formal analysis of dynamic software and hardware systems. In this field, classical approaches usually assume that the system under consideration is static. The states and transitions of the system are collected in an inalterable transition graph (Kripke structure), and the only dynamic aspect is the state change via fixed transitions. How-

ever, from a modern point of view, a system itself is rather dynamic than static; some components in a system may fail while faulty parts may be replaced, or the system may be extended with new components. Modeling this dynamics with classical transition systems is indeed problematic. If a system consists of n components, each of which may be subject to failure, the state space of the Kripke structure rises by a factor of 2^n. When new components can be added again and again, the state space becomes infinite. Using connectivity games we can model the changes of the states of the components by relabeling rules (by identifying a state with a certain label). The replacement of a faulty component corresponds to a restoration of a deactivated node; new components can be added with creation rules. It is also a natural point of view that the components of a system should be connected in order to function properly.

Contributions of this Thesis

This thesis introduces new models for the analysis of dynamic networks in terms of infinite two-player games. The proposed routing and connectivity games are powerful frameworks to model routing and connectivity problems. Besides the initiation of these models, we provide – for each of the considered games – results on the general limitations and algorithmic solutions. To obtain these results, we employ and extend methods from the theory of infinite two-player games. In the following we sketch the results that we establish for each of the game models.

The first chapter deals with sabotage games. We extend the existing model, which is due to van Benthem (2005), and refine the known result that solving sabotage games in the reachability version (and also in various of its variants) is Pspace-complete (Löding and Rohde, 2003a). On the one hand, we study sabotage games with a winning condition expressed in linear temporal logic (LTL). In these games, the graphs have labeled vertices, and Runner has to guarantee that the sequence of labels of the visited vertices fulfills a given LTL formula. We show that LTL sabotage games are still solvable in polynomial space. This result clarifies the complexity of solving sabotage games, since many properties (e.g., reachability, safety) can be expressed in LTL. In particular, we illustrate that the complexity of solving sabotage games is mainly determined by the length of the play,

which is bounded by the number of edges in the graph. However, we show that the winner of a LTL sabotage games needs memory in general, whereas the winner in a reachability sabotage game always wins with a memoryless strategy.

On the other hand, we introduce *randomized sabotage games*, in which the Runner is replaced by a probabilistic player. This reflects the concerns that faults are usually better modeled as random events. In these games, after each of Runner's moves, exactly one available edge is randomly chosen for removal. This turns the sabotage game into a Markov decision process. In this setting solving a randomized sabotage game means to determine whether Runner wins a given game at least with a given probability p. We show that randomized sabotage games are still solvable in polynomial space. We also point to a special case in which solving randomized sabotage games is decidable in linear time. This is the case for the existential fragment; there we ask whether Runner wins with a probability > 0. However, we show that in general solving randomized sabotage games is not easier, even if we restrict the winning probability p for that we ask to a fixed ε-neighborhood. More precisely, given a sabotage game and a probability p, the question of whether Runner wins the randomized sabotage game with a probability $\geq p$ is PSPACE-hard even if the value of p is fixed to an ε-neighborhood (which does not belong to the problem instance). So, turning the Blocker into a probabilistic player does not make the analysis easier (unless one breaks down the problem to the existential fragment).

In the second chapter, we introduce dynamic network routing games. We provide results on the solvability of routing games, which depend not only on the considered winning objective for the routing agent but also on the form of the constraints that restrict the demand agent. In general, the constraints may depend on both the blocked channels and the packets in the network. In this case, we show general limitations of algorithmic solutions, namely that routing games are algorithmically solvable only for the winning condition that requires every packet to be delivered within a given number of turns. For the three other winning objectives considered in this chapter we prove that solving routing games is undecidable.

However, we also analyze simpler versions of the routing game where we ease the constraints. One approach is to require that the possibilities of the demand agent to block edges and to generate packets do not depend on the packets in the network (but only on the currently blocked edges).

This reflects the natural assumption that the demands of the environment and the network clients are not affected by the current network traffic (but possibly by the availability of network connections). For these constraints, to which we refer to as *weak constraints*, routing games are algorithmically solvable for all of the considered winning conditions.

Finally, we introduce an even coarser model of the routing games, which are described by so-called *simple constraints*. In this model, the demand agent has the same possibilities to generate packets and block edges in every turn. For these games, we develop routing algorithms that run locally on each network node. This means that we are able to decompose a winning strategy for the routing agent into local strategies for the single network nodes. We show that, depending on the winning condition, each local routing decision at a node only requires knowledge about a certain local neighborhood of this node.

In the third chapter, we present dynamic network connectivity games. We study the decidability and complexity of solving these games in their reachability version (where Constructor has to establish a connected network) and in their safety version (where Constructor has to guarantee that the network always stays connected). Perhaps surprisingly, our results differ for the safety and the reachability version. Furthermore, the results depend on the types of rules we allow for Constructor. For the most general model, where Constructor's rules may contain any type of rule, we prove that solving connectivity games is undecidable; this holds for both the reachability and the safety version.

However, for several fragments where Constructor's rules are restricted or node labels are omitted, we show that connectivity games are algorithmically solvable. For most of the non-trivial fragments, we show that these games are solvable in Pspace or ExpTime. We also prove the Pspace-hardness of solving connectivity games for many of these cases. Some of the complexity results depend on the balance between node deletion and restoration, i.e., on the fact that Destructor can immediately delete a node after each of Constructor's node creation or restoration. Therefore, we also propose a variant of the model which contains rules that allow Constructor multiple movement and relabeling actions in a single turn. These rules rise the complexity of solving connectivity games in some of the decidable cases; in particular we show that then some of the problems become ExpTime-complete.

Related Work

The starting point for our studies are the aforementioned sabotage games, which were proposed by van Benthem (2005); we will discuss them in detail in the first chapter. The theory of these games and their variants was thoroughly studied by several authors (Löding and Rohde, 2003a,b,c; Rohde, 2004, 2005; Gierasimczuk, Kurzen, and Velázquez-Quesada, 2009; Kurzen, 2011). Besides the complexity results on solving sabotage games in many variants, the mentioned authors developed various sabotage modal logics.

To the best of our knowledge, other contributions are only loosely related to our research on dynamic networks. For the sake of completeness, we mention in the following some other approaches to analyze dynamically changing systems and, especially, dynamic networks. However, most of these approaches do not consider the notion of network modifications by adversarial agents in terms of faults and demands.

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 Altman et al., 2006); these approaches do not consider the evolvement of a network in which the network has to react on faults and demands.

From a more general point of view, dynamically changing systems are also addressed by *online algorithms* (see Fiat and Woeginger, 1998; Borodin and El-Yaniv, 1998; Albers, 2003). These find applications in routing and scheduling problems in wireless and dynamically changing wired networks (see Rajaraman, 2002; Scheideler, 2002). Studies on routing under adversarial demands were started early by Awerbuch, Mansour, and Shavit (1989) and by Borodin, Kleinberg, et al. (1996), but for a long time only the injections of new packets were modeled as adversarial actions. An approach where the adversary also changes the network structure is due to Awerbuch, Berenbrink, et al. (2001) (also see Awerbuch, Brinkmann, and Scheideler, 2003). In their approach, 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*, whose anal-

ysis focuses on the time complexity of update steps to maintain a solution (see Demetrescu et al., 2010; Feigenbaum and Kannan, 2000). In particular, 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 Holm, de Lichtenberg, and Thorup, 2001; Roditty and Zwick, 2004). The same idea motivates studies in the field of *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 Weber and Schwentick, 2007).

Routing problems in dynamic networks were also considered in the theory of *time-varying graphs* (for an overview see Casteigts, Flocchini, Quattrociocchi, et al., 2011). In this field, the edges (or vertices) changes over time; they are described by a "presence function" which defines the time intervals for which edges (or vertices) are available. Also a latency that changes over time is given; it is described by a "latency function" which indicates the time that one needs to cross an edge. So, in contrast to online algorithms, the entire sequence of graphs is provided deterministically in advance, but the model can be used to study routing algorithms whose knowledge about the time-varying graph is restricted. In particular, Casteigts, Flocchini, Mans, et al. (2010) study under what assumptions and knowledge of the time-varying graph one can broadcast information while guaranteeing the "foremost" date of message arrival, the "shortest" routes (i.e., the lowest number of hops), or the "fastest" broadcasting time (i.e., the shortest time spent for broadcasting). Although various stochastic models for time-varying graphs were considered (see Casteigts, Flocchini, Quattrociocchi, et al., 2011), adversarial changes to the graphs were not treated.

A different approach to dynamic systems 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 (Pucella and Weissman, 2004). The logic $\mathrm{DLP}^{+}_{\mathrm{dyn}}$ (see Demri, 2005; Göller and Lohrey, 2006) extends DLP with formulae 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.

There are also models where the dynamics are prvided with the Kripke structure itself. In these structures with *reactive Kripke semantics*, visiting certain nodes or traversing certain edges turns other edges available or unavailable (Gabbay, 2008). But also these models are deterministic and do not consider an adversarial agent.

The idea of changing networks is of course studied in considerable depth in the theory of graph grammars, graph rewriting, and graph transformations (see Rozenberg, 1997; Heckel, 2006). There, the class of generable graphs (networks) is the focus of study, whereas this thesis deals with the more refined view when considering the evolvement of a two-player game and the properties of a (possibly infinite) play.

Finally, in the (one-player) framework of model checking, we mention the work of Gadducci, Heckel, and Koch (1998), 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 apply model checking techniques to dynamically changing Kripke structures directly is *module checking* (Kupferman, Vardi, and Wolper, 2001). There, in the context of open (reactive) 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.

Acknowledgments

Since this thesis is the result of almost five years of my doctoral studies, there are indeed a lot of people to whom I want to express my gratitude. I am deeply indebted to my supervisor Wolfgang Thomas for his support, guidance, and encouragement. He gave me the opportunity to work on an emerging topic which leaves much room for creativity; and, besides his own contributions to my project, he initiated the collaborations with James Gross, Dominik Klein, and Sten Grüner. I am also very grateful to Berthold Vöcking and James Gross, who did not hesitate to act as reviewers for this thesis.

The routing games in this thesis were developed in several fruitful discussions with James Gross. I am very thankful for his devotion to the project and for his patience in making me more sensitive for the practical limitations of our approach. I am equally thankful to Dominik Klein and Sten Grüner, who were excellent diploma students and paved the way to many results on the randomized sabotage games and the connectivity games, respectively. Also, I like to thank Christof Löding for many helpful discussions and, especially, for taking the time to examine some of my proofs very thoroughly. He not only found some flaws in early versions of my results but also provided the ideas to fix them.

I like to give very special thanks to Namit Chaturvedi, Ingo Felscher, Wladimir Fridman, Marcus Gelderie, Sten Grüner, Christof Löding, Daniel Neider, Wied Pakusa, Eva Radmacher, Franz-Josef Radmacher, Stefan Repke, Alexandra Spelten, Guillermo Zecua, and Martin Zimmermann for proofreading drafts of my thesis. Many thanks are also given to Michael Ummels, Stefan Repke, and the "Aachener TEX-Stammtisch" for their help and suggestions in typesetting this thesis. I am also very grateful to all my colleagues from the "i7 & MGI" crew (the past as well as present members) for the moments we shared; they not only inspired me but also made my time in Aachen very enjoyable. Finally, I like to thank my family and my friends for their encouragement, support, and patience.

Runner vs. Blocker

1

Sabotage Games & Randomization

Our studies on games for dynamically changing networks have their origin in the *sabotage game*, which was proposed by van Benthem in 2002.[1] A sabotage game is played on a graph between two players, called *Runner* and *Blocker*. In the original setting, Runner has to reach a given set of final vertices while, after each of Runner's moves, Blocker removes an edge from the graph. But already van Benthem (2005) mentioned a lot of variants of this game in his original article, e.g., a traveling salesman problem with sabotage, a three-player game in that two Runners try to meet or avoid each other while Blocker deletes edges, and a probabilistic version where Blocker is replaced by a random player. Other studied variants comprise safety sabotage games, in which Runner has to avoid some vertex set (Gierasimczuk, Kurzen, and Velázquez-Quesada, 2009), sabotage games in which Blocker deletes multiple edges per turn or deletes vertices (Löding and Rohde, 2003a), and sabotage games with restorations (Rohde, 2005).

1 His article was published three years later (van Benthem, 2005).

In this chapter, we consider two different kinds of winning objectives for Runner. The first one is a reachability winning condition, i.e., as in van Benthem's original article, Runner has to reach a given vertex set. The other one is a more general winning condition stated in linear temporal logic (LTL); there, the vertices of the graph are labeled and Runner has to guarantee that the sequence of labels of the visited vertices fulfills a given LTL formula. We introduce reachability and LTL sabotage games more formally in Section 1.1.

There are several reasons for considering sabotage games with the less general reachability objective. First, the hardness results for solving sabotage games are already valid in this simple setting. Second, strategies do not need memory, i.e., the players only need to know the current position of the game to make an optimal move. And third, a simple reachability condition provides an easy approach to sabotage games and gives a good intuition for the game-theoretic nature of routing problems. With the LTL winning objective on the other hand, we study a powerful winning condition which comprises many of the aforementioned winning conditions (e.g., reachability and safety).

Besides the standard model of the sabotage game, where Runner plays against a malicious Blocker, we consider also a randomized version where the edge deletions happen randomly, i.e., after each of Runner's moves, exactly one edge is deleted randomly. We define these randomized sabotage games in detail in Section 1.2.

Löding and Rohde (2003a,b) already showed that solving reachability sabotage games is PSPACE-complete. In this chapter, we clarify and extend this result. On the one hand, we provide a deterministic algorithm for solving sabotage games that requires only polynomial space. We provide two versions of the algorithm, one for reachability and one for LTL sabotage games. Our algorithmic solution provides also a solution for the randomized sabotage games in polynomial space. In this case, the algorithm returns the maximal winning probability with which Runner can win the given game. Moreover, our algorithm can be used to compute the strategy for the winner of the game. For reachability sabotage games the obtained strategy is always positional (memoryless), whereas memory is generally required to win an LTL sabotage game (Section 1.3).

On the other hand, we extend the hardness result from Löding and Rohde (2003a,b), which we present in Section 1.4, to randomized sabotage

games. At first sight this seems to be clear, since a solution for winning a randomized sabotage game surely, i.e., with probability 1, is identical to a solution for the two-player version of this game. Opposed to this, the question of whether Runner wins the randomized version of a reachability sabotage game with a probability > 0 is decidable in linear time. Therefore, we restrict Runner's winning probability p for which we ask to a fixed ε-neighborhood. For any fixed ε-neighborhood we show the following: Given a sabotage game and a p in this ε-neighborhood, the question of whether Runner wins the randomized sabotage game with a probability $\geq p$ is PSPACE-hard (i.e., the probability p of the problem instance is restricted to an a priory fixed ε-neighborhood). The proof of this result consists of two steps. The first one is a reduction from solving randomized sabotage games for a parametrized probability to solving two-player sabotage games (Section 1.5); this allows us to use the known hardness result for the two-player version. In a second step, we show that we can choose the parameters for the reduction in such a way that the parametrized probability lies in the desired ε-neighborhood; and we show that we find these parameters in polynomial time (Section 1.6).

The hardness result for randomized sabotage games, which we show in Sections 1.5 and 1.6, has been developed in discussion and collaboration with Dominik Klein and Wolfgang Thomas (see Klein, Radmacher, and Thomas, 2009). The generalization to sabotage games with LTL specifications was originally published in an extended version of the aforementioned work (see Klein, Radmacher, and Thomas, 2012).

1.1 The Sabotage Game

Van Benthem (2005) introduced the sabotage game as a turn-based two-player game. It is played on a given finite graph in the way that a *Runner* traverses the graph while an adversary, *Blocker*, deletes an edge after each turn. Since the given graph is finite, the game ends after Blocker has deleted all edges.

The given graph can be either directed or undirected, and the edges may possibly be multi-edges. In this thesis, we define a *graph* as well as a *multi graph* as a pair (V, E), where V is a non-empty, finite set of vertices. If G is a (usual) graph, which only contains single edges, E is an

edge relation $E \subseteq V \times V$. If G is a multi graph, E is an edge multiplicity function $V \times V \to \mathbb{N}$, where $\mathbb{N} = \{0, 1, 2, \ldots\}$ denotes the set of natural numbers including zero. In the latter case, we also write $(u, v) \in E$ if $E(u, v) > 0$. Two vertices u and v are *adjacent* if $(u, v) \in E$. A graph is *undirected* if $(u, v) \in E$ implies $(v, u) \in E$. A multi graph is *undirected* if $E(u, v) = E(v, u)$ for all vertices u and v. Otherwise the (multi) graph is *directed*. We denote with $|E| \in \mathbb{N}$ the number of single edges of $\mathcal{G}$. For directed graphs with single edges, this corresponds to the standard definition of the number of pairs in the edge relation E. For directed graphs with multi-edges, we set $|E| := \sum_{e \in V \times V} E(e)$. For undirected graphs we divide these values by two, so that we count each edge only once.

A *sabotage game* is a pair

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

consisting of a graph or multi graph $G = (V, E)$ and a designated *initial vertex* $v_{\text{in}} \in V$. A *position* of the game is a pair (v_n, E_n) with $v_n \in V$ and $E_n \subseteq E$. The *initial position* is (v_{in}, E). In each turn of the game (where, say, (v_n, E_n) is the current position), first Runner chooses an outgoing edge (v_n, v_{n+1}) from vertex v_n and moves to vertex v_{n+1}. After Runner's move, the new position of the game is (v_{n+1}, E_n); we say that Runner has *reached* the vertex v_{n+1}. Then, Blocker chooses an edge $e \in E$. If G is a (usual) graph, we remove e and define $E_{n+1} := E_n \setminus \{e\}$. If G is a multi graph, we decrement the multiplicity of e by one; we define E_{n+1} by $E_{n+1}(e) := E_n(e) - 1$ and $E_{n+1}(e') := E_n(e')$ for all $e' \neq e$. Of course, if G is an undirected graph and Blocker removes the edge $e = (u, v)$, we also remove the pair (v, u) from E_n or reduce the multiplicity of (v, u) by one. After Blocker's move, the turn is over; the new position of the game is (v_{n+1}, E_{n+1}).

A *play* of the sabotage game is a sequence of positions

$$\pi = (v_0, E_0)(v_1, E_0)(v_1, E_1) \cdots (v_m, E_m)$$

where $(v_0, E_0) = (v_{\text{in}}, E)$ is the initial position, each step from (v_n, E_n) to (v_{n+1}, E_n) results from a move of Runner, and each step from (v_n, E_{n+1}) to (v_{n+1}, E_{n+1}) results from an edge deletion of Blocker. A play ends if Runner cannot move anymore, i.e., there does not exist any outgoing edge from Runner's current vertex. Clearly, the number of reachable positions

of a game is finite. And since the players are not permitted to skip and edges are only deleted and not added, also each play is finite.

In this work, we consider two kinds of winning conditions. The first one is a *reachability winning condition* where Runner's objective is to reach a given set of *final* (or *goal*) vertices $F \subseteq V$. The other is a more general *LTL winning condition* which is given by an LTL formula φ and a labeling $L\colon V \to 2^{AP}$ of the vertices with atomic propositions from a given set AP; Runner's aim is to traverse the game graph in such a way that the sequence of labels of the visited vertices fulfills the given LTL formula φ. Although the LTL condition is more general than the reachability condition, we introduce both conditions since we show our PSPACE-hardness result even for the simpler reachability winning condition.

Linear Temporal Logic (LTL)

In the following we recall the syntax and semantics of LTL (cf. Clarke, Grumberg, and Peled, 2000; Baier and Katoen, 2008). The syntax of LTL formulae over a set AP of atomic propositions is defined as follows:

- $\texttt{true}$ is an LTL formula.
- Each atomic proposition in AP is an LTL formula.
- If φ_1 and φ_2 are LTL formulae, then $\neg\varphi_1$, $\varphi_1 \wedge \varphi_2$, $\mathrm{X}\,\varphi_1$, and $\varphi_1 \,\mathrm{U}\, \varphi_2$ are LTL formulae.

As usual we define $\texttt{false} := \neg\texttt{true}$, $\varphi_1 \vee \varphi_2 := \neg(\neg\varphi_1 \wedge \neg\varphi_2)$, $\mathrm{F}\,\varphi_1 := \texttt{true}\,\mathrm{U}\,\varphi_1$, and $\mathrm{G}\,\varphi_1 := \neg\,\mathrm{F}\neg\varphi_1$ for LTL formulae φ_1, φ_2.

We define the semantics of LTL over a path labeled with atomic propositions in AP. For a path $\rho = v_0 v_1 \cdots v_{n-1}$, let $\rho[i] = v_i$ and $\rho[i..]$ be the path $v_i v_{i+1} \cdots v_{n-1}$. Given a path $\rho \in V^*$, a labeling function $L\colon V \to 2^{AP}$, and an LTL formula φ, the satisfaction relation $\models_L$ is defined inductively as follows:

$$
\begin{array}{lll}
\rho \models_L \texttt{true} & & \\
\rho \models_L a & \text{iff} & a \in L(\rho[0]) \\
\rho \models_L \neg\varphi & \text{iff} & \rho \not\models_L \varphi \\
\rho \models_L \varphi_1 \wedge \varphi_2 & \text{iff} & \rho \models_L \varphi_1 \text{ and } \rho \models_L \varphi_2
\end{array}
$$

$$\begin{aligned}
\rho \vDash_L \mathrm{X}\,\varphi \quad &\text{iff} \quad \rho[1..] \vDash_L \varphi \\
\rho \vDash_L \varphi_1 \,\mathrm{U}\, \varphi_2 \quad &\text{iff} \quad \exists j \geq 0 \colon \rho[j..] \vDash_L \varphi_2 \text{ and} \\
&\qquad\quad \forall 0 \leq k < j \colon \rho[k..] \vDash_L \varphi_1 \,.
\end{aligned}$$

Winning Conditions

In order to evaluate a play of a sabotage game, we define for a play $\pi = (v_0, E_0)(v_1, E_0)(v_1, E_1)\cdots(v_m, E_m)$ the *trace of* π as

$$\mathrm{trace}(\pi) := v_0 v_1 \cdots v_m \,,$$

i.e., we project the play to the sequence of vertices visited by Runner.

We can consider a sabotage game $\mathcal{G} = (G, v_{\mathrm{in}})$ either as a *reachability sabotage game* with a set F of final vertices or as an *LTL sabotage game* with an LTL specification φ and a labeling $L \colon V \to 2^{AP}$ of the vertices with atomic propositions from a given set AP. We say that *Runner wins a play* π *of the reachability sabotage game* $\mathcal{G}$ if $\mathrm{trace}(\pi)$ contains a vertex from the set F; Blocker wins otherwise. Analogously, *Runner wins a play* π *of the LTL sabotage game* $\mathcal{G}$ if $\mathrm{trace}(\pi) \vDash_L \varphi$; otherwise Blocker wins.

Clearly, the LTL winning condition is more general, since each reachability objective can be expressed by an LTL formula without changing the game graph, but not conversely. To transform a reachability sabotage game into an LTL sabotage game, take the LTL formula $\mathrm{F}\, a$ and label exactly the final vertices with a.

Strategies and Determinacy

For the formal analysis of sabotage games, we introduce the notion of strategies. A *strategy for Runner* is a (partial) function $\sigma \colon (V \times 2^E)^+ \to V$; it maps each play prefix $(v_0, E_0)(v_1, E_0)(v_1, E_1)\cdots(v_n, E_n)$ to a vertex v_{n+1} with $(v_n, v_{n+1}) \in E_n$ (meaning that Runner moves to v_{n+1} if she plays according to σ). A *strategy for Blocker* is a (partial) function $\tau \colon (V \times 2^E)^+ \to E_{n-1}$; it maps each play prefix $(v_0, E_0)(v_1, E_0)(v_1, E_1)\cdots(v_n, E_{n-1})$ to the edge which Blocker removes next. A strategy for Runner is a *finite-memory* strategy if it only depends on the current position and on the state of a finite-memory structure (e.g., a finite-state automaton) that is maintained according to the visited vertices, i.e., there exists a finite set S such that the

strategy can be described as a function $\sigma\colon V \times 2^E \times S \to V$, which maps Runner's current position (v_n, E_n) and the current memory state $s \in S$ to the vertex v_{n+1}. Moreover, a strategy for Runner is *positional* (or *memoryless*) if it only depends on the current position, i.e., it is a function $\sigma\colon V \times 2^E \to V$, which maps Runner's current position (v_n, E_n) to the vertex v_{n+1}. For Blocker, finite-memory and positional strategies are defined analogously. *Runner wins the reachability (LTL) game* if she has a strategy σ to win every play of the reachability (LTL) game in which she moves according to σ. Analogously, Blocker wins the reachability (LTL) game if he has a strategy τ to win every play. We call a strategy with which a particular player wins a *winning strategy* for this player.

It is easy to show that every sabotage game with a reachability or an LTL winning condition is *determined*, i.e., in each sabotage game either Runner or Blocker has a winning strategy. This follows from a classical result of Martin (1975), namely that every game with a Borel type winning condition is determined (also see Martin, 1985; Kechris, 1995). To apply this result, it suffices to transform the sabotage game into a *two-player game played on a graph* (see Thomas, 2008a; Grädel, Thomas, and Wilke, 2002), in which each vertex corresponds exactly to one position of the sabotage game and an indication of which player acts next. The resulting game graph has vertices of the form (i, v_n, E_n) containing the player i who moves next and the position (v_n, E_n) of the sabotage game; the winning condition defined on the vertices of the sabotage game carries over to the new unfolded game graph.[2]

In the same way one can show that positional strategies are sufficient to win reachability sabotage games, i.e., if one of the players wins a reachability sabotage game $\mathcal{G}$, this player also has a positional strategy to win $\mathcal{G}$. This follows since in the unfolded reachability game the winning player can always win with a positional strategy (see Thomas, 1995, 2008a; Grädel, Thomas, and Wilke, 2002).

Proposition 1.1. *Sabotage games with a Borel type winning condition over the visited vertices (e.g., reachability or LTL sabotage games) are determined. Moreover, in reachability sabotage games the winning player always has a positional winning strategy.*

2 Rohde (2005) states this unfolding explicitly for reachability sabotage games.

The Problem of Solving Sabotage Games

The task that we address in this chapter is to solve a given sabotage game. Usually, a solution of a sabotage game $\mathcal{G}$ comprises the winner of $\mathcal{G}$ and a winning strategy. To classify this problem in terms of computational complexity (see Papadimitriou, 1994), we formally only refer to the question of whether Runner wins $\mathcal{G}$. Thus, the problem of solving sabotage games denotes one of the following decision problems.

- *Solving reachability sabotage games:* Given a reachability sabotage game $\mathcal{G} = ((V, E), v_{\text{in}})$ with a designated set $F \subseteq V$, does Runner win the reachability sabotage game (i.e., does Runner have a strategy to reach a vertex in F)?

- *Solving LTL sabotage games:* Given a sabotage game $\mathcal{G} = ((V, E), v_{\text{in}})$, a labeling $L\colon V \to 2^{AP}$ of the vertices, and an LTL formula φ, does Runner win the LTL sabotage game (i.e., does Runner have a strategy to traverse the graph in a way that the sequence of labels of the visited vertices fulfills the LTL formula φ)?

In the face of analyzing the computational complexity of these decision problems, we have to define more precisely in which format an instance is given. All ingredients of the graph are given in a unary coding (also edge multiplicities are coded unary), the labeling of the vertices are given as bit vectors over the atomic propositions, and the LTL formula is given as a tree where each node represents a subformula. Thus, the size of a problem instance of a reachability sabotage game is $|V| + |E| + |F|$, and the size of a problem instance of an LTL sabotage game is $|V| + |E| + |V| \cdot |AP| + |\operatorname{cl}(\varphi)|$, where $|E|$ is the number of single edges as defined before and $\operatorname{cl}(\varphi)$ is the set of all subformulae of φ.

In practice, beyond determining the winner of a sabotage game, we want to synthesize a winning strategy for the player who wins. For example, for the synthesis of a routing algorithm that has to function in a scenario modeled by a sabotage game, we want to obtain a strategy for Runner. For the formal verification of systems, a counter-example for an unsatisfiable specification is helpful, which in terms of sabotage games corresponds to a winning strategy for Blocker. The algorithms for solving sabotage games that we shall present in Section 1.3 can easily be modified to compute a

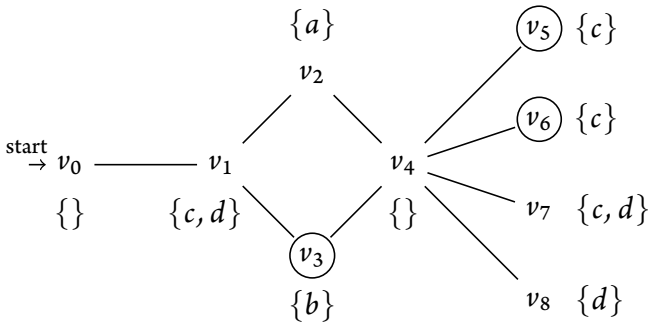

Figure 1.1: The game graph of a sabotage game. In the reachability game, Runner has to reach the set of circled final vertices; in the LTL game, the sequence of labels of the visited vertices over the set $AP = \{a, b, c, d\}$ has to fulfill a given LTL formula.

winning strategy for the winning player. Moreover, for reachability sabotage games we can compute in this way a positional winning strategy. For an LTL sabotage game we only obtain a finite-memory winning strategy. The following example shows that, in general, positional strategies are not sufficient to win LTL sabotage games.

Example 1.1. As a comprehensive example we consider the sabotage game in Figure 1.1. The depicted graph is undirected, and it only contains single edges. The vertex v_0 is the initial vertex.

First, we analyze the reachability sabotage game where Runner (starting in the initial vertex) has to reach one of the circled final vertices. It is easy to see that Blocker wins this game. In the first turn Runner has to move to v_1. Then, Blocker has to remove the edge (v_1, v_3) in order to prevent that Runner wins. In the second turn Runner either moves to v_2 or back to v_1. In either case, Blocker will delete the edge (v_2, v_4) in his following move; thus, Runner is cut off from any final vertex. Hence, Blocker wins.

Next, we imagine the variant where the vertices v_2 and v_4 are connected by a double edge. In the resulting reachability sabotage game, the situation is essentially better for Runner. Again, Runner starts by moving to v_1, and Blocker has to remove the edge (v_1, v_3) in his first move in order to prevent Runner from winning. In the second turn Blocker can delete only one of the two edges between v_2 and v_4; thus, he cannot prevent Runner from reaching the vertex v_4. Since Blocker can remove only two of the

edges (v_4, v_3), (v_4, v_5), and (v_4, v_6) in the second and third turn, Runner is able to move via v_4 to one of the final vertices v_3, v_5, and v_6. Hence, Runner wins in this scenario.

Now, we consider the LTL sabotage game where the labeling of the graph over the set $AP = \{a, b, c, d\}$ is depicted in Figure 1.1 and the given LTL formula is

$$\varphi := \mathrm{F}(a \wedge \mathrm{F}\, c) \vee \mathrm{F}(b \wedge \mathrm{F}\, d)\,.$$

Runner has to traverse the game graph in such a way that the sequence of visited vertices fulfills φ. In fact, Runner wins with the following strategy. In the first turn Runner moves to v_1; in the second turn Runner moves to v_3 if the edge (v_1, v_2) or the edge (v_2, v_4) is deleted, and Runner moves to v_2 otherwise. At v_2 (v_3) Runner moves back to v_1 if the edge (v_1, v_2) (the edge (v_1, v_3)) is available and to v_4 otherwise. At v_4 Runner moves to v_5,v_6, or v_7 if she has visited v_2, and she moves to v_7 or v_8 otherwise.

Note that Runner does not have a *positional* strategy to win this LTL sabotage game. Assume the following two sequences of edge deletions: (v_1, v_3), (v_1, v_2), (v_4, v_7) and (v_1, v_2), (v_1, v_3), (v_4, v_7). We neglect that it is possible for Runner to move at v_1 back to vertex v_0, because Runner cannot win with such a strategy. So, both sequences lead to the same position with Runner at vertex v_4; but the first sequence forces Runner to move via v_2 while the second forces Runner to move via v_3. Then, at v_4, Runner has to move either to one of the vertices v_5, v_6 or to the vertex v_8 depending on the sequence of previous edge deletions.

1.2 The Randomized Sabotage Game

Since Blocker's edge deletions usually represents failures in a system, his actions are in many scenarios better modeled as random events. Therefore, we introduce in this section the modified game model where Blocker is replaced by a purely probabilistic adversary, to which we often refer as *Nature*. With the term *Nature* we follow Papadimitriou (1985, 1994), who denotes such games, where a single player is faced with a probabilistic adversary, as *games against nature*. Depending on the community, the terms *Markov decision process* (see Filar and Vrieze, 1996; Puterman, 2005; Baier and Katoen, 2008) or *1½-player game* (see Chatterjee and Henzinger, 2012) are probably more common in the literature.

As before in the two-player version, a *randomized sabotage game* is denoted by a pair $\mathcal{G} = (G, v_{\text{in}})$ consisting of a graph $G = (V, E)$ and an initial vertex $v_{\text{in}} \in V$. We describe a *position* of the game again by a pair (v_n, E_n), but we do not collect the intermediate positions where Nature acts. In a turn of a randomized sabotage game, first Runner chooses an outgoing edge (v_n, v_{n+1}) in vertex v_n of position (v_n, E_n) and moves to vertex v_{n+1}. Then, Nature acts by removing randomly a single edge. For this, a dice with $|E_n|$ sides is thrown, and the chosen edge is removed from E_n. Note that we defined $|E_n| = \sum_{e \in V \times V} E_n(e)$ for multi graphs. So, in a multi graph Nature only reduces the multiplicity of an edge by one. The new edge relation (or edge multiplicity function) E_{n+1} is defined accordingly as in the two-player version. After this turn, the new position of the game is (v_{n+1}, E_{n+1}); we say that Runner has *reached* the vertex v_{n+1}. So, a *play* of a randomized sabotage game is a sequence of positions

$$\pi = (v_0, E_0)(v_1, E_1) \cdots (v_m, E_m)$$

where $(v_0, E_0) = (v_{\text{in}}, E)$ and each step from (v_n, E_n) to (v_{n+1}, E_{n+1}) results from a move of Runner followed by an edge deletion. For such a play π we define the *trace of* π as $\text{trace}(\pi) := v_0 v_1 \cdots v_m$.

Evaluating Randomized Sabotage Games

For the probabilistic analysis of randomized sabotage games we need the formal definition of a *strategy for Runner*: analogously to the two-player game, this is a (partial) function $\sigma\colon (V \times 2^E)^+ \to V$ which maps each play prefix $(v_0, E_0)(v_1, E_1) \cdots (v_n, E_n)$ to a vertex v_{n+1} with $(v_n, v_{n+1}) \in E_n$ (meaning that Runner moves to v_{n+1} if she plays according to σ).

Now we build up the *probability tree* $t_{\mathcal{G},\sigma}$ for a randomized sabotage game $\mathcal{G}$ and a strategy σ. The probability tree contains the positions (of Runner) that may occur when Runner plays according to the strategy σ. The root of the probability tree is (v_{in}, E). From a position (v', E') the successor nodes are all positions (v'', E'') where v'' is the vertex $\sigma(v', E')$ and E'' results from E' by an edge deletion. So, a position where Runner cannot move (and where thus $\sigma(v', E')$ is not defined) is a leaf.

Given the probability tree $t_{\mathcal{G},\sigma}$ and the winning condition of $\mathcal{G}$ (which is either a set F of final vertices for a reachability game or an LTL formula φ for an LTL game with a labeling L), we define the *set of winning plays* as

the set of paths from the root to a leaf in the probability tree where Runner wins according to the winning condition. Formally, for a reachability and an LTL winning condition we define the *set of winning plays* as follows:

$$\begin{aligned}\mathrm{Plays}(t_{\mathcal{G},\sigma}, F) &:= \text{set of all paths } \pi \text{ from the root to a leaf in}\\ &\qquad t_{\mathcal{G},\sigma} \text{ where } \mathrm{trace}(\pi) \text{ contains a vertex in } F\,;\\ \mathrm{Plays}(t_{\mathcal{G},\sigma}, (L, \varphi)) &:= \text{set of all paths } \pi \text{ from the root to a leaf in}\\ &\qquad t_{\mathcal{G},\sigma} \text{ with } \mathrm{trace}(\pi) \vDash_L \varphi\,.\end{aligned}$$

For a play $\pi = (v_0, E_0)(v_1, E_1)\cdots(v_n, E_n)$ in one of these sets, we denote with $\mathrm{Prob}_{t_{\mathcal{G},\sigma}}(\pi)$ the *probability of* π, i.e.,

$$\mathrm{Prob}_{t_{\mathcal{G},\sigma}}(\pi) := \frac{1}{|E_0|} \cdot \frac{1}{|E_1|} \cdots \frac{1}{|E_{n-1}|}\,.$$

Given a randomized sabotage game $\mathcal{G}$, we say that *Runner wins the reachability game with probability* p (*Runner wins the LTL game with probability* p) if she has a strategy σ such that the probabilities of all plays in $\mathrm{Plays}(t_{\mathcal{G},\sigma}, F)$ (in $\mathrm{Plays}(t_{\mathcal{G},\sigma}, (L, \varphi))$) sum up to p, i.e., if

$$p = \sum_{\pi \in W} \mathrm{Prob}_{t_{\mathcal{G},\sigma}}(\pi)\,,$$

where $W = \mathrm{Plays}(t_{\mathcal{G},\sigma}, F)$ in the case of a reachability game and where $W = \mathrm{Plays}(t_{\mathcal{G},\sigma}, (L, \varphi))$ in the case of an LTL game.

The Problem of Solving Randomized Sabotage Games

A solution of a randomized sabotage game $\mathcal{G}$ usually comprises the maximal probability p for which Runner wins $\mathcal{G}$ and a strategy for Runner to win with probability p. Again, to classify this problem in terms of computational complexity (see Papadimitriou, 1994), we formally refer to the decision problem of whether, for given $\mathcal{G}$ and p, Runner wins $\mathcal{G}$ at least with a probability $\geq p$. Thus, formally the problem of solving randomized sabotage games denotes one of the following decision problems.

- *Solving randomized reachability sabotage games:* Given a randomized sabotage game $\mathcal{G} = ((V, E), v_{\mathrm{in}})$, a designated set $F \subseteq V$, and a $p \in [0, 1]$, does Runner win the reachability sabotage game with a probability $\geq p$?

- *Solving randomized LTL sabotage games:* Given a randomized sabotage game $\mathcal{G} = ((V, E), v_{\text{in}})$, a labeling $L: V \to 2^{AP}$ of the vertices, an LTL formula φ, and a $p \in [0, 1]$, does Runner win the LTL sabotage game with a probability $\geq p$?

The algorithms that we present in Section 1.3 for solving randomized sabotage games also return Runner's winning probability for a given randomized sabotage game $\mathcal{G}$. For this reason, the probability p can be given in any representation (e.g., binary) as long as we can efficiently compare p with the obtained winning probability for Runner. Besides Runner's winning probability our algorithms can easily be modified to compute an optimal[3] strategy for Runner.

Also note that the problem of solving two-player sabotage games, as introduced in the previous section, is a special case of solving randomized sabotage games: Runner wins the two-player reachability (LTL) sabotage game if and only if she wins the randomized reachability (LTL) sabotage game with probability 1.

Example 1.2. Let us consider again the game graph in Figure 1.1. First, we analyze the randomized reachability sabotage game where Runner (starting in the initial vertex) has to reach one of the circled final vertices. We ask whether Runner wins the reachability game with a probability ≥ 0.95. We have already seen in Example 1.1 that Runner cannot guarantee to reach one of the final vertices surely, i.e., with probability 1. However, it is easy to state a strategy for Runner which maximizes her winning probability. In the first turn Runner has to move to v_1. In the second turn Runner tries to move to the final vertex v_3 if the edge (v_1, v_3) has not been deleted, and she moves to v_2 otherwise. At v_2 she moves to v_4 if possible. If Runner reaches v_4, she moves to one of the final vertices v_3, v_5, v_6 via a non-deleted edge. However, it may happen with a probability > 0 that after Runner's first move the edge (v_1, v_3) is deleted (for which the probability is $\frac{1}{9}$) and that after her second move the edge (v_2, v_4) is deleted (for which the probability is $\frac{1}{8}$). So, Runner loses the game with a probability of $\frac{1}{72}$, and hence she wins with a probability of $\frac{71}{72} > 0.98$, which is higher than the probability of 0.95 we asked for.

3 A strategy for Runner is *optimal* if it maximizes Runner's winning probability.

Now, we consider the randomized LTL sabotage game (where the game graph and its labeling over the set $AP = \{a, b, c, d\}$ are defined again as in Figure 1.1). As specification we provide the LTL formula

$$\varphi := \mathrm{X}\,\mathrm{X}\,\neg b \wedge \mathrm{X}\,\mathrm{X}\,\mathrm{X}\,\mathrm{X}\,\neg b\,,$$

i.e., a safety condition where Runner has to avoid to visit a b-labeled vertex in the second and the fourth turn. We ask whether Runner wins this LTL sabotage game with a probability ≥ 0.95. In this game Runner has to make four moves without visiting v_3. Clearly, Runner wins if she is able to move via the vertex v_2 to v_4 (since after three turns there is at least one edge left that does not lead to v_3). Let us calculate Runner's maximal winning probability. For this we distinguish four cases, depending on the first edge that is removed by Nature. If in the first turn none of the edges (v_0, v_1), (v_1, v_2), (v_2, v_4) is deleted, Runner wins surely: Runner moves to v_2, and even if then the edge (v_2, v_4) fails, Runner moves back to v_1; in her fourth move she can avoid v_3 by moving to v_0 or v_2. So, in this case Runner can maximize her winning probability by playing the strategy where she always tries to avoid the vertex v_3 and tries to move via v_2 to v_4. This case, namely that one of the three edges is deleted after Runner's first move, happens with probability $\frac{6}{9}$. If in the first turn (v_0, v_1) is deleted, Runner only loses if (v_2, v_4) and then (v_1, v_2) are deleted in the following turns (which happens with probability $\frac{1}{8} \cdot \frac{1}{7}$). If in the first turn (v_1, v_2) is deleted, Runner only loses if (v_0, v_1), (v_1, v_2) stay intact in the next turn and (v_0, v_1) is deleted in the following turn (which happens with probability $\frac{6}{8} \cdot \frac{1}{7}$). If in the first turn (v_2, v_4) is deleted and Runner moves to v_2 in her second move, she only loses if (v_0, v_1) and then (v_1, v_2) are deleted in the following turns (which happens with probability $\frac{1}{8} \cdot \frac{1}{7}$). Hence, Runner wins with a probability of

$$\frac{6}{9} + \frac{2}{9}\left(1 - \frac{1}{8} \cdot \frac{1}{7}\right) + \frac{1}{9}\left(1 - \frac{6}{8} \cdot \frac{1}{7}\right) = \frac{62}{63} > 0.98\,,$$

which is higher than the probability of 0.95 we asked for.

Anyway, calculating the winning probability for Runner can be a hard task even for simple LTL formulae. As such an example one can consider the randomize LTL sabotage game on the same graph, but with the specification $\neg\,\mathrm{F}\,b$, i.e., the safety game where Runner has to avoid to visit

vertex v_3 as long as she is able to move. Since in this case we cannot abort the computation after four turns, it would be troublesome to compute by hand whether Runner still wins with a probability ≥ 0.95.

1.3 Solving Sabotage Games in Polynomial Space

Löding and Rohde (2003a,b) showed that solving two-player sabotage games is PSPACE-complete for various winning conditions: reachability, *Hamilton path* (i.e., Runner wins if she visits each vertex exactly once), and *complete search* (i.e., Runner wins if she visits each vertex at least once). Rohde (2005) shows the membership of these problems to PSPACE by providing an alternating polynomial-time algorithm. Such an algorithm guesses Runner's moves non-deterministically and chooses the blocked edges universally, i.e., the algorithm has to accept for all possible edge deletions. The membership of the problem in PSPACE follows from APTIME = PSPACE, where we denote with APTIME the class of decision problems that can solved by an *alternating Turing machine* in polynomial time (see Chandra, Kozen, and Stockmeyer, 1981; Papadimitriou, 1994).

It is known that replacing universal quantifiers by randomized quantifiers in a quantified Boolean formula does not lead out of the class PSPACE for deciding whether such a formula is satisfiable (Papadimitriou, 1985; Littman, Majercik, and Pitassi, 2001). So, it is no surprise that the randomized versions of these sabotage games are also solvable in PSPACE.

In contrast to the LTL model checking problems for two-player games (Pnueli and Rosner, 1989; Rosner, 1991; Alur, La Torre, and Madhusudan, 2003) and Markov decision processes (Courcoubetis and Yannakakis, 1995) on static graphs, which have both been shown to be complete for double exponential time, we can also solve (randomized) LTL sabotage games in PSPACE. The reason is that the number of turns of every sabotage game is bounded by the number of edges of the initial graph.

In this section we provide algorithms that solve (randomized) sabotage games in PSPACE. This clarifies the algorithmic solution of randomized sabotage games. In the randomized case we cannot adapt the alternating polynomial-time algorithm from Rohde (2005) since we cannot choose randomly deleted edges universally. However, we still have to inspect each possible edge deletion to compute Runner's winning probability. So, our

algorithm can be seen as a deterministic variant of Rohde's alternating algorithm. More precisely, we generate the game tree on-the-fly in a depth-first manner. This allows us to memorize Runner's winning probability in the currently inspected subgame for each possible edge deletion.

Furthermore, our algorithmic solution shows that for every randomized reachability sabotage game Runner always has a positional strategy that maximizes her winning probability and that for every randomized LTL sabotage game Runner has a finite-memory strategy that is optimal in this sense. For this reason, we treat the solution of reachability and LTL sabotage games separately in this section. We only provide algorithms for the randomized versions since the algorithms can be used for the two-player version by checking whether Runner's winning probability equals 1. At the end of the section we discuss how the algorithms can be adapted for solving further variants of sabotage games.

1.3.1 The Reachability Case

In the following we provide an algorithm that computes, for a given reachability sabotage game (with a given set of final vertices), the winning probability for Runner. The space that this algorithm uses is only polynomial in the size of the given game instance (which we have defined as the number of vertices and edges of the given graph plus the size of the set of final vertices). So, by comparing the result of this algorithm to a given probability p one can decide in polynomial space whether Runner wins a randomized reachability sabotage game at least with probability p.

We present the Algorithm 1.1, which builds up and traverses the game tree in a depth-first manner. More precisely, for a reachability sabotage game $\mathcal{G} = (G_{\text{in}}, v_{\text{in}})$ with a set F of final vertices, the *game tree* consists of both all reachable positions where it is Runner's turn and all reachable intermediate positions where Nature acts next. Each tree node has the form $(G, v, 0/1)$ consisting of the current game graph G, Runner's current position v, and a bit which is 1 if and only if Runner moves next. The root of the game tree is $(G_{\text{in}}, v_{\text{in}}, 1)$, where Runner starts to move. From a position $(G, v, 1)$ with $G = (V, E)$ and $v \notin F$, where Runner moves, the successor nodes are all positions $(G, v', 0)$ with $(v, v') \in E$ (a position (v, E), with $v \in F$ or $(v, v') \notin E$ for all v', is a leaf). Now, the successors of $(G, v', 0)$ are the positions $(G', v', 1)$ with $G' = (V, E')$ where E' re-

Algorithm 1.1: An algorithm for solving (randomized) reachability sabotage games.

Input: the game graph G, current position u of Runner (initialized to the initial vertex), the set F of final vertices, boolean variable *runners-turn* (initialized to 1)

Output: Runner's winning probability p_{out} in the game (G, u) for reaching a final vertex in F

Function: `solve-reach-game`$(G, u, F, \textit{runners-turn})$

```
1  if u ∈ F then
2    p_out := 1
3  else
4    if u has no outgoing edges then
5      p_out := 0
6    else (* u has at least one outgoing edge *)
7      if runners-turn then (* Runner moves next *)
8        p_out := 0
9        let m be the number of outgoing edges from u
10       for i := 1 to m do
11         let (u, v_i) be the i-th outgoing edge from u
12         p_out := max{p_out, solve-reach-game(G, v_i, F, 0)}
13       end for
14     else (* Nature acts next *)
15       p_out := 0
16       let n be the total number of edges in G
17       for j := 1 to n do
18         let G_j be the game graph resulting from G by deletion of
               the j-th edge e_j (i.e., decrementing its multiplicity
           by 1)
19         p_out := p_out + (1/n) · solve-reach-game(G_j, u, F, 1)
20       end for
21     end if
22   end if
23 end if
24 return p_out
```

sults from E by an edge deletion (i.e., by decrementing an edge multiplicity by 1 if G is a multi graph).

To each node of Runner our algorithm assigns the probability for Runner to win the subgame starting in this node. These probabilities are computed inductively in the obvious way, starting with 1 and 0 at a leaf $(G, v, 1)$ depending on whether v is in F or not. For an inner node s in the game tree which has successors $s_1, \ldots, s_k$ with winning probabilities p_i $(1 \leq i \leq k)$, the algorithms assigns to s the winning probability p_{out} as follows. If s is a tree node of Nature, then $p_{\text{out}} := \frac{1}{k} \sum_i p_i$ (each edge is hit with the same probability). If s is a tree node of Runner, $p_{\text{out}} := \max_i p_i$ (Runner moves to the vertex with the maximal winning probability). Thus, Runner wins $\mathcal{G}$ with the probability p_{out} assigned to the root node.

So, using Algorithm 1.1 we obtain Runner's winning probability p_{out} for the reachability game by calling the function

$$\texttt{solve-reach-game}(G, v_{\text{in}}, F, 1)\,.$$

Since the algorithm builds up and traverses the entire game tree, the computation may require exponential time in general. However, the required space is still polynomial. The memory that is needed for each recursive call is linear in $|G| := |V| + |E|$.[4] Since the depth of the game tree and hence the recursion depth of the function `solve-reach-game` is bounded by the number of edges $|E|$, the algorithm runs in $O(|G|^2)$ space.

Hence, given a randomized reachability sabotage game and p, one can decide in PSPACE whether Runner wins with a probability $\geq p$. For that, one calls the function `solve-reach-game` and compares the output value p_{out} with p. The algorithm can also be used for deciding whether Runner wins a two-player reachability sabotage game since this is exactly the case when Runner wins in the randomized version with probability 1.

We can also adapt Algorithm 1.1 to compute an optimal strategy for Runner that is positional. For this, in the for loop for Runner's move (lines 10–13), where the current game graph is $G = (V, E)$ and Runner is at vertex u, we have to memorize the v_i for which the recursive call of the function $\texttt{solve-reach-game}(G, v_i, F, 0)$ maximizes the probability p_{out}.

4 We store the probability p_{out} in binary encoding. The current value arises by taking at most $|E|$ times the maximum and the arithmetic mean of the successor nodes (starting from 0 and 1 at the leaves). So, the memory needed to store p_{out} is linear in $|E|$.

Then, an optimal positional strategy for Runner is the following: When the play is in position (E, u), Runner moves to vertex v_i. Since this strategy per definition maximizes Runner's winning probability, Runner always has an optimal, positional strategy. Especially, this means that Runner cannot gain a higher winning probability with a randomized strategy. We summarize our results on solving reachability sabotage games with the following theorem.

Theorem 1.2. *Solving (randomized) reachability sabotage games is in* PSPACE*, i.e., given a (randomized) reachability sabotage game $\mathcal{G}$ with a goal set F (and a $p \in [0, 1]$), one can decide in polynomial space whether Runner wins $\mathcal{G}$ (with a probability $\geq p$). Moreover, given a randomized reachability sabotage game $\mathcal{G}$ with a goal set F, one can compute in polynomial space Runner's winning probability for $\mathcal{G}$ and a positional strategy that maximizes her winning probability.*

1.3.2 The LTL Case

In the following we provide an algorithm that computes the winning probability for Runner in a given LTL sabotage game (for a given LTL formula and a given labeled graph). The most obvious approach for solving an LTL sabotage game might be to traverse the game tree in a depth-first manner and to compute the probabilities of each subformula at each node (as it works for reachability sabotage games for a single reachability property). However, this approach does not work since one choice may maximize the probability for fulfilling one subformula whereas another choice may maximize the probability for fulfilling another subformula.

However, the height of the game tree is bounded by the number of edges in the game graph. So, we can compute at each leaf in polynomial time whether the path that Runner has taken satisfies the given LTL formula. This idea is implemented in Algorithm 1.2; during traversing the game tree it stores the *history* of Runner's moves, i.e., the sequence of vertices that Runner has visited in a play. The winner of a play is computed at each leaf of the game tree by the function `play-satisfies-formula`, which returns true if and only if the sequence of labelings of the play satisfies the LTL formula, i.e.,

$$\texttt{play-satisfies-formula}(\rho, L, \varphi) := \begin{cases} \texttt{true} & \text{if } \rho \vDash_L \varphi \\ \texttt{false} & \text{otherwise}. \end{cases}$$

Then again, for each node of the game tree where Nature acts next, we take the arithmetic mean over the successors; for a node where Runner moves next we take the maximal probability of the successors. Runner wins the LTL sabotage game with the probability p_{out} that is assigned to the root node.

So, given a randomized sabotage game $\mathcal{G} = (G, v_{\text{in}})$ over an graph $G = (V, E)$, a labeling function $L\colon V \to 2^{AP}$, and an LTL formula φ, the call of the function

$$\texttt{solve-LTL-game}(G, v_{\text{in}}, L, \varphi, 1)$$

of Algorithm 1.2 returns Runner's winning probability p_{out} for the LTL game $\mathcal{G}$. For the complexity analysis of the algorithm, we first note that the additional memory that is needed per call is linear in $|V|+|E|$ if the last node of the current play prefix is an inner node of the game tree (either one node is added to the history ρ or one edge in $|E|$ is removed). For a leaf the function `play-satisfies-formula` is called, which requires at most $O(|E|\cdot|\operatorname{cl}(\varphi)|)$ space, where $\operatorname{cl}(\varphi)$ denotes the set of all subformulae of φ. Since the depth of the game tree and hence the recursion depth of `solve-LTL-game` is bounded by the number of edges $|E|$, the algorithm requires the space

$$O\left(|E|\cdot(|V|+|E|)+|E|\cdot|\operatorname{cl}(\varphi)|\right) \subseteq O\left(|G|^2+|E|\cdot|\operatorname{cl}(\varphi)|\right).$$

It remains to be shown that the function `play-satisfies-formula` requires at most $O(|E|\cdot|\operatorname{cl}(\varphi)|)$ space. More precisely, for a history $\rho = u_1\cdots u_n$, the function `play-satisfies-formula`(ρ, L, φ) runs in time

$$O(n\cdot|\operatorname{cl}(\varphi)|\cdot(n+|AP|))$$

and in $O(n\cdot|\operatorname{cl}(\varphi)|)$ space. This can be done by filling a matrix with $|\operatorname{cl}(\varphi)|$ rows and n columns; each (i, j)-entry of this matrix is a bit which is set to 1 if and only if $\rho[j..]$ satisfies the i-th subformula. The computation of each matrix entry is in $O(|AP|+n)$ time and constant space: For an atomic proposition, we have to inspect at most a bit vector of size $|AP|$ (i.e., we have check whether the atomic proposition hold at the vertex v_i); for all other entries, we have to inspect at most two other rows of the matrix (this is the case for an until formula).

Algorithm 1.2: An algorithm for solving (randomized) LTL sabotage games.

Input: the game graph G, the history ρ of Runner's moves including his current position (initialized to the initial vertex), the LTL formula φ, boolean variable *runners-turn* (initialized to 1)

Output: Runner's winning probability p_{out} for satisfying φ in the original game under the assumption that she played ρ and reaches G

Function: `solve-LTL-game`$(G, \rho, L, \varphi,$ *runners-turn*$)$

let u be Runner's current position after playing ρ, i.e., $\rho = \rho' u$ for some ρ'
if *u has no outgoing edges* **then**
  if `play-satisfies-formula`(ρ, L, φ) **then**
    $p_{\text{out}} := 1$
  else
    $p_{\text{out}} := 0$
  end if
else $(*$ *u has at least one outgoing edge* $*)$
  if *runners-turn* **then** $(*$ *Runner moves next* $*)$
    $p_{\text{out}} := 0$
    let m be the number of outgoing edges from u
    for $i := 1$ to m **do**
      let (u, v_i) be the i-th outgoing edge from u
      $p_{\text{out}} := \max\{p_{\text{out}},$ `solve-LTL-game`$(G, \rho v_i, L, \varphi, 0)\}$
    end for
  else $(*$ *Nature acts next* $*)$
    $p_{\text{out}} := 0$
    let n be the total number of edges in G
    for $j := 1$ to n **do**
      let G_j be the game graph resulting from G by deletion of the j-th edge e_j (i.e., decrementing its multiplicity by 1)
      $p_{\text{out}} := p_{\text{out}} + \frac{1}{n} \cdot$ `solve-LTL-game`$(G_j, \rho, L, \varphi, 1)$
    end for
  end if
end if
return p_{out}

Hence, given a randomized LTL sabotage game and p, one can decide in PSPACE whether Runner wins with a probability $\geq p$. For that, one calls `solve-LTL-game` and compares the output value p_{out} with p. The algorithm can also be used for deciding whether Runner wins a two-player LTL sabotage game since this is exactly the case when Runner wins in the randomized version with probability 1.

We have already seen in Example 1.2 that positional strategies, in general, do not suffice to win LTL sabotage games. Nevertheless, we can adapt Algorithm 1.2 to compute an optimal strategy for Runner that only needs finite memory. For that, in the for loop for Runner's move (lines 12–15), where the current game graph is $G = (V, E)$ and the history of the play is $\rho = u_1 \cdots u_n$, we have to memorize the v_i for which the recursive call `solve-LTL-game`$(G, \rho v_i, L, \varphi, 0)$ maximizes the probability p_{out}. We can store the strategy by assigning to the pair $((E, u_n), \rho)$ of position and history the vertex v_i (for which p_{out} is maximal). Then, an optimal finite-memory strategy for Runner is the following: When the play is in position (E, u_n) after Runner has moved according to the history ρ, Runner moves to vertex v_i. Since the length of a sequence ρ, on which the strategy depends, is at most $|E|$, finite-memory strategies are sufficient to maximize the winning probability in LTL sabotage games. Especially, this means that Runner cannot gain a higher winning probability with a randomized strategy. We summarize our results on solving LTL sabotage games with the following theorem.

Theorem 1.3. *Solving (randomized) LTL sabotage games is in PSPACE, i.e., given a (randomized) LTL sabotage game $\mathcal{G}$ with an labeling function L and an LTL formula φ (and a $p \in [0, 1]$), one can decide in polynomial space whether Runner wins $\mathcal{G}$ (with a probability $\geq p$). Moreover, given a randomized LTL sabotage game $\mathcal{G}$ with L and φ, one can compute in polynomial space Runner's winning probability for $\mathcal{G}$ and a positional strategy that maximizes her winning probability.*

1.3.3 Further Results

To get a complete picture we briefly mention some possible extensions of the presented algorithms to solve some variants and generalizations of the sabotage game. On the one hand, we remark that Algorithm 1.2 for solving

sabotage games with an LTL winning condition only checks at each leaf of the game tree whether the generated path satisfies the given LTL formula. By replacing the function `play-satisfies-formula`, the algorithm can be adapted to check more general specifications of linear time properties; e.g., these could be given by regular expressions or in the extended temporal logic ETL (Vardi and Wolper, 1994). Indeed, for every linear time property for that we can check in polynomial space whether it is fulfilled by a given sequence of labels, the corresponding randomized sabotage game can also be solved in PSPACE.[5] For example, the winning conditions *complete search* and *Hamilton path* (see Löding and Rohde, 2003a,b) can be checked in this way also in their randomized version. More precisely, one has to check whether the generated path contains each vertex of the game graph at least once for the complete search condition or exactly once for the Hamilton path condition.

We constricted our algorithms to a uniform probability distribution of edge failures and the assumption that Nature deletes exactly one edge per turn. We can adapt both Algorithm 1.1 and Algorithm 1.2 for solving more general versions of sabotage games where the probabilities for edge failures are non-uniform and where also more than one edge may fail in each turn. For this, we need to modify our algorithms as follows.

1. For solving randomized sabotage games with non-uniform distributions of edge failures, let us assume that for each position (v, E) of the game a distribution of edge failures $p_{v,E}(e) \mapsto [0, 1]$ with $\sum_{e \in E} p_{v,E}(e) = 1$ is given. Then, in line 19 of Algorithm 1.1 (and in line 21 of Algorithm 1.2) we just have to replace the factor $\frac{1}{n}$ by $p_{u,E}(e_j)$ (where E describes the edges in the graph G before the deletion of e_j).

2. For solving randomized sabotage games in which k edges fail in each turn, it suffices to replace the boolean variable *runners-turn* by a modulo-$(k + 1)$ counter (so that Nature deletes k edges in succession). With some work our algorithms can also be adapted to the case where in each turn the number of edge deletions is chosen randomly (in $\{1, \ldots, k\}$). For that, the algorithm has to calculate

5 For instance one can check efficiently whether the sequence of labels matches a regular expression (see Aho, 1990).

the probability for all k cases recursively and take the arithmetic mean of these probabilities. However, we have to require that at least one edge is deleted per turn; otherwise we cannot guarantee the termination of our algorithms.

1.4 PSPACE-Hardness of the Reachability Game

So far, we have seen that (randomized) sabotage games can be solved in polynomial space; this holds for sabotage games with various winning conditions (e.g., reachability, Hamilton path, complete search, LTL, regular expressions) on graphs which may be directed or undirected and may possibly contain multi-edges. Löding and Rohde (2003a,b) have already shown that solving sabotage games with a reachability, a Hamilton path, or a complete search winning condition is also Pspace-hard. In Section 1.5 we will refine the hardness result in the probabilistic setting. In order to have a self-contained exposition this section briefly describes the hardness proof by Löding and Rohde for solving two-player reachability sabotage games.

We show the hardness by a reduction from the problem *Quantified Boolean Formulae (QBF)*, which is known to be Pspace-complete (see Papadimitriou, 1994, problem *QSAT*). The basic strategy is to construct a game graph in such a way that, in the first part of the game graph, Runner chooses the assignments for existentially quantified variables and Blocker chooses the assignments for the universally quantified variables. Then, these assignments are verified in a second part.

Formally, an instance of the problem QBF is given as a quantified boolean formula in a special form; more precisely: given a boolean expression ϑ in conjunctive normal form over boolean variables $x_1, \ldots, x_m$, is

$$\exists x_1 \forall x_2 \exists x_3 \ldots Q_m x_m \; \vartheta$$

true? Without loss of generality, one requires the formula to start with an existential quantifier. If m is even, $Q_m = \forall$; otherwise $Q_m = \exists$. We also assume that each clause of ϑ contains exactly three literals. For each instance φ of QBF, we construct a reachability sabotage game $\mathcal{G}$ with a set F of final vertices such that φ is true if and only if Runner has a strategy to reach a final vertex in $\mathcal{G}$.

The game graph that we construct consists of three types of subgraphs or *gadgets*: existential, universal, and verification gadgets. The gadgets uses multi-edges. Here, we denote an edge with multiplicity l as *l-edge*. Although, the gadgets also contain *∞-edges*; these can be considered as permanent edges, which cannot be removed by Runner. Formally, ∞-edges are only syntactic sugar since in two-player reachability sabotage games they can be replaced by l-edges for a sufficiently large l. More precisely, the following is easy to show: If Runner wins a two-player reachability sabotage game, she has also a winning strategy without visiting any vertex twice (see Rohde, 2005). This implies that the multiplicity of the edges can be bounded to the number of vertices in the graph. (Moreover, for the constructions in this chapter, it suffices to consider ∞-edges as 4-edges. Nevertheless, we think that the construction with ∞-edges is more convenient for the reader.) In the following we describe these different gadgets. At the end of this section, we introduce an additional gadget with which we can replace multi-edges with single edges; this shows that solving reachability sabotage games is already PSPACE-hard on graphs that only contain single edges.

The Existential Gadget

Intuitively, the existential gadget allows Runner to set an existentially quantified variable to true or false. The gadget is depicted in Figure 1.2. Runner's aim is to traverse the gadget, i.e., to move from vertex a to b, in such a way that the 4-edge between x_i and the (circled) final vertex stays intact if she sets the variable x_i to true and the 4-edge between $\overline{x_i}$ and the final vertex stays intact if she sets the variable x_i to false. We denote the vertex a as the entry vertex, the vertices x_i and $\overline{x_i}$ as the variable vertices, and the vertex b as the exit vertex of this gadget. If this is the first existential gadget (i.e., $i = 1$), the entry vertex is the initial vertex of the constructed game; otherwise, the entry vertex coincides with the exit vertex of the preceding gadget (i.e., the universal gadget for x_{i-1}). Analogously, the exit vertex of this gadget coincides with the entry vertex of the next gadget (i.e., the universal gadget for x_{i+1}, or the verification gadget if x_i is the last quantified variable). The "back"-edges from x_i and $\overline{x_i}$ lead directly to the last gadget of the construction, the verification gadget. Later, Runner possibly moves back via these edges to verify her assignment of the variable x_i. (We will

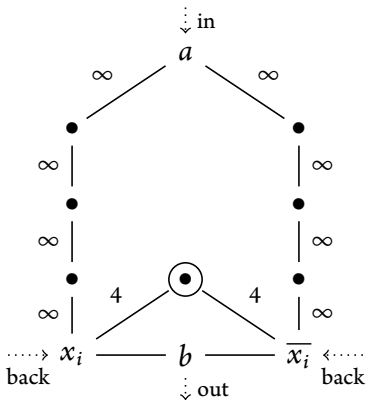

Figure 1.2: The existential gadget for the variable x_i (if i is odd).

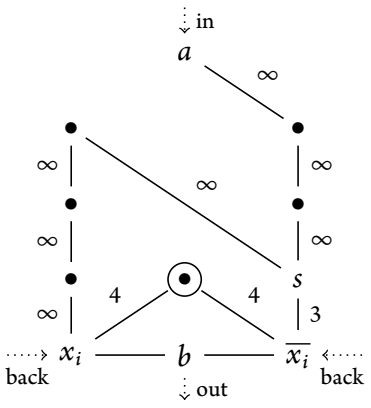

Figure 1.3: The universal gadget for the variable x_i (if i is even).

see later that it is useless for Runner to take these edges as a shortcut from the existential gadget directly to the verification gadget.)

If Runner wants to set x_i to true, she moves along the right path to the vertex $\overline{x_i}$ in four turns. In this case Blocker successively removes the 4-edge between $\overline{x_i}$ and the final vertex to prevent Runner from winning immediately. When in the subsequent turn Runner moves to b, Blocker removes the edge between b and x_i to prevent Runner from reaching the final vertex via x_i. So, after Runner traverses the gadget the 4-edge from x_i to the final vertex remains untouched so far while all edges between $\overline{x_i}$ to the final vertex are removed. The case that Runner wants to set x_i to false is symmetrically. Hence, if Runner later moves from the verification gadget back to x_i or $\overline{x_i}$, she can only guarantee a win from one of these vertices; and Runner chooses which case applies.

The Universal Gadget

The universal gadget lets Blocker choose the truth value of a universally quantified variable. The gadget is depicted in Figure 1.3. Runner should be able to traverse the gadget from a to b, but Blocker can decide whether the 4-edge between x_i and the final vertex or the 4-edge between $\overline{x_i}$ and the final vertex stays intact. We denote vertices as entry, variable, or exit vertex in the same way as in the existential gadget, and the exit vertex b coincides

with the entry vertex of the next gadget (i.e., the existential gadget for x_{i+1}, or the verification gadget if x_i is the last quantified variable). Again, the "back"-edges from x_i and $\overline{x_i}$ lead directly to the verification gadget and possibly Runner moves back via these edges to verify Blocker's assignment of the variable x_i. (Also in the universal gadget it is useless for Runner to take these edges as a shortcut directly to the verification gadget.)

If Blocker wants to set x_i to false, he removes the three edges between s and $\overline{x_i}$. Runner can only guarantee to leave the gadget at b via x_i, but she cannot visit $\overline{x_i}$ or the (circled) final vertex because Blocker removes the 4-edge from x_i to the final vertex and the edge between b and $\overline{x_i}$ in the subsequent turns. In this case, however, the 4-edge between $\overline{x_i}$ and the final vertex remains untouched.

If Blocker wants to set x_i to true, he removes successively the 4-edge from $\overline{x_i}$ to the final vertex. At s Runner will move down to $\overline{x_i}$, because in this way Runner can achieve that the 4-edge between x_i and the final vertex stays untouched when she reaches b. (If Runner misbehaves in s by moving via the ∞-edges and x_i to the exit vertex b, Blocker can completely delete both 4-edges to the final vertex.)

Hence, Blocker chooses which 4-edge to the final vertex stays intact and which is completely removed. If Runner later moves back to x_i or $\overline{x_i}$ from the verification gadget, Blocker can only win if Blocker has assigned the corresponding truth value to the variable x_i before.

The Verification Gadget

The verification gadget can be seen as an evaluation game for the quantifier-free part of the quantified Boolean formula, where Blocker picks a clause and Runner picks a literal of this clause. Then, Runner has to move back to the corresponding variable vertex (in an existential or universal gadget) and wins there if the 4-edge to the final vertex has not been removed before. Thus, Runner wins if the assignments for the variables, which have been chosen in the existential and universal gadgets, satisfy the quantifier-free part of the formula.

The verification gadget for a formula with k clauses $C_1, \dots, C_k$ is depicted in Figure 1.4. Its entry vertex coincides with the exit vertex b of the last of the existential or universal gadgets. For a clause $C_i = (\neg)x_{i_1} \vee (\neg)x_{i_2} \vee (\neg)x_{i_3}$ there are three paths. Each of these paths leads from c_i

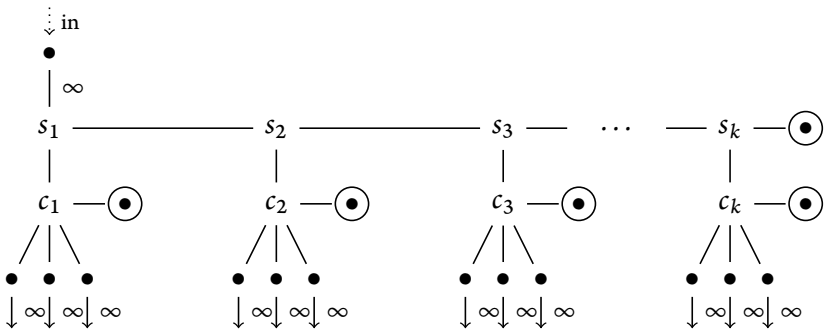

Figure 1.4: The verification gadget for a formula with k clauses.

via a single edge and an ∞-edge back to the variable vertex x_{i_j} ($\overline{x_{i_j}}$) in the corresponding gadgets, where $j \in \{1, 2, 3\}$ denotes the first, second, or third path. We denote the corresponding edge as the literal edge L_{ij}.

Runner's first turn in the verification gadget leads her to the vertex s_1. Whenever Runner is at vertex s_i, Blocker can decide whether Runner can move to c_i or to s_{i+1}. If, however, Runner is at s_k, Runner can proceed to c_k since Blocker has to remove the edge between s_k and the final vertex. So, Blocker decides to which of the vertices $c_1, \ldots, c_k$, which represent the clauses $C_1, \ldots, C_k$, Runner will move. When Runner moves to such a vertex c_i, Blocker has to remove the edge between c_i and the final vertex. Then, Runner can move via one of the three literal edges L_{ij} back to the variable vertex x_{i_j} ($\overline{x_{i_j}}$) if Runner or Blocker has set x_i to true (false) in the corresponding existential or universal gadget. So, Runner chooses one of the three literals $(\neg)x_{i_1}$, $(\neg)x_{i_2}$, $(\neg)x_{i_3}$ in the clause C_i. Runner wins by moving from the chosen variable vertex x_{i_j} ($\overline{x_{i_j}}$) to the final vertex if and only if the 4-edge has not been removed before.

One should note that Runner cannot win by moving from an existential or universal gadget via a "back"-edge (i.e., a literal edge L_{ij}) to the verification gadget since Blocker will then remove the edge between c_i and the literal edge L_{ij}. The construction contains, however, many edges to prevent Runner from traversing edges a non-intended way. The gadgets can be simplified if edges are directed (Löding and Rohde, 2003a).

It is now easy to see that Runner has a winning strategy in the constructed reachability sabotage game if and only if the quantified Boolean

formula φ is true. If φ is true, Runner is able to choose assignments of the variables in the existential gadgets against all assignments of Blocker in the universal gadgets such that these assignments satisfy the quantifier-free part of φ. Then, from each vertex c_i in the verification gadget there exists a path via a literal edge back to a final vertex in an existential or universal gadget because each clause has a literal which is true for the chosen assignments. So, Runner wins by moving along this path. Conversely, if φ is false, Blocker is able to choose assignments of the variables in the universal gadgets against all assignments of Runner in the existential gadgets such that the quantifier-free part of φ if false with these assignments. Then, Blocker forces Runner to move to a vertex c_i, which represents a clause C_i that is false for the chosen assignments. There, Runner gets stuck since each of the three paths back to the previous gadgets lead to a variable vertex that is disconnected from the final vertex.

As a final remark we note that the number of vertices and edges of the constructed sabotage game is linear in the length of the given quantified Boolean formula. The size of each existential and each universal gadget is fixed; the size of the verification gadget is linear in the length of the formula. This linear time bound requires that each ∞-edge is replaced by an edge with a constant multiplicity, e.g., the construction still works if each ∞-edge is replaced by a 4-edge. However, we do not prove this linear time bound. Therefore, let us also note that a polynomial bound for the size of the constructed game (which also meets our need) is easy to obtain. For this we set the multiplicity of each ∞-edge to the total number of vertices in all gadgets. This generic bound follows from the aforementioned fact that in a two-player sabotage game Runner has a strategy to win without visiting any vertex twice if she wins at all (see Rohde, 2005). In any case, we can use the construction for a reduction from the problem QBF. Since the problem QBF is PSPACE-hard, the same holds for solving sabotage games.

On Graphs with Single Edges or a Single Goal

All of the presented gadgets contain multi-edges. Thus, one might question whether multi-edges are needed for obtaining the presented hardness result. Löding and Rohde (2003a,b) showed that solving reachability sabotage games stays PSPACE-hard under this restriction since each l-edge

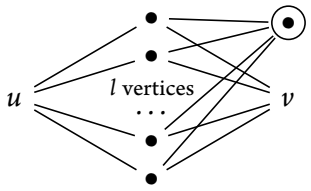

Figure 1.5: An l-edge from u to v.

from a vertex u to a vertex v can be replaced by the construction shown in Figure 1.5, which only involves single edges. (The figure shows the replacement for undirected graphs; the construction for directed graphs is the same except that all edges are directed from left to right.) The resulting game graph is equivalent in the sense that Runner wins the reachability sabotage game on the original graph if and only if she wins the reachability game on the modified graph where every multi-edge is replaced; the size of the resulting game graph is polynomial in the size of the original game graph. So, solving reachability sabotage games remains PSPACE-hard on game graphs with single edges.

Another question is whether multiple goal vertices are needed for the presented reduction from QBF. Indeed, we can merge all final vertices into one vertex if we allow multi-edges. So, solving sabotage games remains PSPACE-hard on graphs containing only one final vertex. In this case, however, we cannot replace all multi-edges by single edges in general. If for instance in Figure 1.5 the vertex v is also final, merging the final vertices leads again to multi-edges (more precisely, to l double edges). In general, if multi-edges are forbidden, we can only reduce the number of final vertices to two (Löding and Rohde, 2003a,b).

We summarize the hardness-results of this section.

Theorem 1.4 (Löding and Rohde, 2003a,b)**.** *Solving reachability sabotage games is PSPACE-hard, i.e., the following problem is PSPACE-hard: Given a sabotage game $\mathcal{G} = (G, v_{in})$ with a goal set F, does Runner have a strategy to win $\mathcal{G}$? Moreover, this problem remains PSPACE-hard if*

- *F is restricted to be a singleton set and the multiplicity of each edge in G is ≤ 2, or*
- *F is restricted to contain only two vertices and G has only single edges.*

If the given game graph has only single edges and only one final vertex, we can solve reachability sabotage games in linear time (Löding and Rohde, 2003a). The observation that leads to this result is that Runner wins such a reachability sabotage game if and only if she either starts at the final vertex or reaches the final vertex in her first move; if Runner moves to another vertex v (which is not the goal), Blocker deletes the edge between v and the goal vertex (if there is any).

1.5 PSPACE-Hardness of the Randomized Reachability Game for Arbitrary Probabilities

In this section we extend the PSPACE-hardness result of Löding and Rohde to randomized sabotage games. Indeed, solving randomized sabotage games is PSPACE-hard in general because it is already PSPACE-hard to decide whether Runner wins a reachability sabotage game with probability 1, which is equivalent to the question of whether Runner wins the two-player sabotage game. Nevertheless, the problem may become easier if we restrict the probability for which we ask to a certain probability. For instance, the problem of whether Runner wins a randomized reachability sabotage game with a probability > 0 is decidable in linear time since Runner wins with a probability > 0 if and only if there is a path from the initial to a final vertex. We shall show, however, that solving randomized reachability sabotage games remains PSPACE-hard if we restrict the probability to any ε-neighborhood. More precisely, for any fixed $p \in [0, 1]$ and $\varepsilon > 0$, the problem of whether Runner wins a randomized reachability game at least with a given probability p' is PSPACE-hard, even if p' is restricted to the interval $[p - \varepsilon, p + \varepsilon]$.[6]

For the proof of this result we use a parametrized reduction from solving two-player reachability sabotage games, whose PSPACE-hardness we discussed in the last section. For each reachability sabotage game $\mathcal{G}$, we construct a family of instances (with natural numbers k and n as parameters), each of which consisting of a randomized reachability sabotage game $\mathcal{G}_{k,n}$ and a rational number $p_{k,n} \in [0, 1]$. The construction ensures that,

6 So, for any $\varepsilon > 0$, asking whether Runner wins with a probability $\geq p$ with $p \in [0; \varepsilon]$ instead of asking whether Runner wins with a probability > 0 rises the computational complexity of solving randomized sabotage games from linear time to PSPACE.

for each k and n, Runner wins the two-player game $\mathcal{G}$ exactly if Runner wins the randomized game $\mathcal{G}_{k,n}$ with a probability $\geq p_{k,n}$. Furthermore, we guarantee that, given p and $\varepsilon > 0$, the probability $p_{k,n}$ lies within the interval $[p - \varepsilon, p + \varepsilon]$ for suitable k and n, and that such k and n can be computed in polynomial time.

The outline of the proof is as follows. We first introduce the construction of the randomized sabotage game $\mathcal{G}_{k,n}$ and show that Runner wins $\mathcal{G}_{k,n}$ with probability $p_{k,n}$ if and only if Runner wins the given two-player sabotage game $\mathcal{G}$ (Section 1.5.1). In a second step, we exactly calculate the probability $p_{k,n}$, with which Runner wins $\mathcal{G}_{k,n}$ (Section 1.5.2). To complete the proof, we finally show that the set of the probabilities $p_{k,n}$ is dense in the interval $[0, 1]$, and that the parameters k and n can be computed efficiently such that $p_{k,n} \in [p - \varepsilon, p + \varepsilon]$ (Section 1.5.3).

1.5.1 The Randomized Sabotage Game $\mathcal{G}_{k,n}$

The intention of the parameters is that a higher k decreases Runner's winning probability, whereas a lower k increases Runner's winning probability; and opposed to this, a higher n increases Runner's winning probability, whereas a lower n decreases Runner's winning probability. Parameter k determines a new subgraph of the constructed game graph, which we denote as the *parametrization gadget* $\mathcal{H}_k$. The parameter n is the overall number of edges in the constructed game graph that do not belong to the parametrization gadget. This leads us the additional constraint that n has to be greater or equal to the number of edges in $\mathcal{G}$; the parameter k could be any natural number ≥ 1.

In detail, let us assume that the graph of the given sabotage game $\mathcal{G}$ has n_0 edges. We get modifications of $\mathcal{G}$ with any number $n \geq n_0$ of edges by adding artificial extra edges (without manipulating the winner of the two-player game); we can achieve this by adding a path (with $n - n_0$ edges) that has a dead end. We call this sabotage game $\mathcal{G}^n$.

Now, we combine $\mathcal{G}^n$ with the parametrization gadget $\mathcal{H}_k$. We define the parametrization gadget $\mathcal{H}_k$ for each $k \geq 1$ as depicted in Figure 1.6. In the depicted version edges are undirected; a version of the parametrization gadget for directed graphs can be obtained by directing each horizontal edge from left to right and each vertical edge from top to bottom. The parametrization gadget is the initial part of the game that we construct,

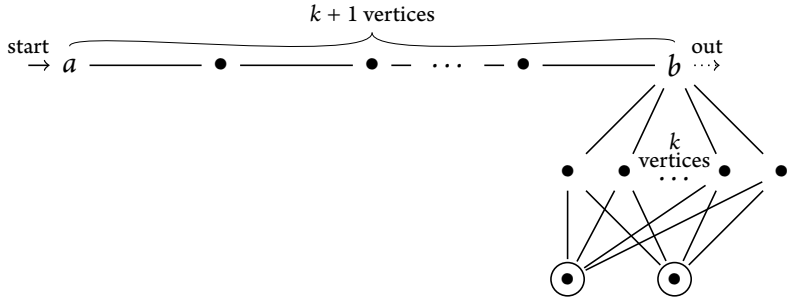

Figure 1.6: The parametrization gadget $\mathcal{H}_k$.

i.e., Runner starts from the initial vertex a in this gadget. We attach the graph of the game $\mathcal{G}^n$ by identifying the exit vertex b in $\mathcal{H}_k$ with the initial vertex of $\mathcal{G}^n$; the resulting game is the randomized sabotage game

$$\mathcal{G}_{k,n} := \mathcal{H}_k \circ \mathcal{G}^n ,$$

with which we will work in the following. We define

$$p_{k,n} := \text{the probability for Runner to traverse the parametrization gadget } \mathcal{H}_k \text{ of } \mathcal{G}_{k,n} ,$$

i.e., the probability for Runner starting at a to reach the vertex b. We show that Runner wins the constructed game $\mathcal{G}_{k,n}$ with probability $p_{k,n}$, i.e., with her probability for traversing the parametrization gadget $\mathcal{H}_k$, if she wins the original two-player sabotage game $\mathcal{G}$. If Runner does not win the original two-player sabotage game $\mathcal{G}$, she only wins $\mathcal{G}_{k,n}$ with a probability that is strictly smaller than $p_{k,n}$.

Lemma 1.5. *Runner wins the randomized reachability sabotage game $\mathcal{G}_{k,n}$ with a probability $\geq p_{k,n}$ iff Runner wins the two-player reachability sabotage game $\mathcal{G}$.*

Proof. First note the following. If Runner resides at vertex b and in $\mathcal{H}_k$ exactly the k edges below vertex b have been deleted so far, then Runner wins with probability 1 iff Runner wins the two-player sabotage game $\mathcal{G}$. The reason is that Runner is at the initial vertex of the original game $\mathcal{G}$, whose edges are untouched except of the additionally added extra edges (which are per definition useless for Runner in the two-player game), and that Runner cannot win from b in the parametrization gadget $\mathcal{H}_k$ anymore.

Now assume that Runner wins the two-player sabotage game $\mathcal{G}$. In the parametrization gadget Runner starts at vertex a and moves towards b. Clearly, if any edge between his current position and b is deleted, she loses the game immediately. However, if Runner succeeds in getting to b, she wins surely from there. To show this, we first distinguish two cases. If during Runner's k steps towards b only edges in $\mathcal{H}_k$ were subject to deletion, Runner wins from b by moving in the game graph of $\mathcal{G}$ with the strategy of the two-player sabotage game $\mathcal{G}$. If otherwise at least one of the k deletions took place outside of $\mathcal{H}_k$, there is at least one edge leading from b downward to some middle node, say b', and from there are at least two edges leading to the two final vertices of the parametrization gadget. Hence, in this case Runner wins from b by moving downwards to b' and then to one of the two final vertices (depending on the next edge deletion). In both cases, Runner wins over $\mathcal{G}_{k,n}$ exactly with the probability $p_{k,n}$ of traversing the parametrization gadget $\mathcal{H}_k$ from node a to node b.

Conversely, assume now that Blocker wins the two-player sabotage game $\mathcal{G}$. In the case that only the k edges below vertex b in $\mathcal{H}_k$ are subject to deletion while Runner moves towards b, which happens with a probability > 0, Runner's only chance to win from b is by moving towards some final vertex in the game graph of $\mathcal{G}$. Since Blocker wins the two-player sabotage game $\mathcal{G}$, Runner's winning probability at b is strictly smaller than 1. Hence, Runner's overall probability to win $\mathcal{G}_{k,n}$ (from the initial vertex a) is strictly smaller than $p_{k,n}$. □

In order to use the previous lemma as a reduction to show PSPACE-hardness, we need to construct the game $\mathcal{G}_{k,n}$ in polynomial time. Since $\mathcal{G}^n$ can be obtained from a given sabotage game $\mathcal{G}$ by adding a path with a dead end of at most n edges, the size of $\mathcal{G}^n$ is linear in n and in the size of $\mathcal{G}$. Concerning the parametrization gadget $\mathcal{H}_k$ it suffices to note that $\mathcal{H}_k$ has $2k + 3$ vertices and $4k$ edges.

Lemma 1.6. *The number of vertices (edges) in $\mathcal{G}_{k,n}$ are linear in k, n, and the number of vertices (edges) in the game graph of the given sabotage game $\mathcal{G}$.*

1.5.2 Computing the Probability $p_{k,n}$

In the following we compute the probabilities $p_{k,n}$ with which Runner succeeds in traversing the parametrization gadget $\mathcal{H}_k$ of a randomized

sabotage game $\mathcal{G}_{k,n}$. If $k = 1$, Runner reaches the vertex b in the first turn; thus, we have $p_{1,n} = 1$ for all n.

For $k \geq 2$, we note that Runner starting at a needs k moves to reach the vertex b, and that Nature may prevent Runner from reaching b only by edge deletions after each of Runner's first $k-1$ moves. So, we obtain Runner's winning probability from the probability of Runner not failing in the first turn, multiplied by the probability of not failing in the second turn, and so on, until the probability of not failing in the $(k-1)$-th turn. If in the first $k-1$ turns Nature has not deleted any edge between Runner's current position and the vertex b, Runner reaches b with the next move (so, Runner cannot fail in the k-th turn).

After Runner's first move she still has to cross $k-1$ edges, none of which may be deleted in Nature's first turn. Since the overall number of edges in the game graph of $\mathcal{G}_{k,n}$ is $4k+n$, Runner does not lose in the first turn if any of the other $4k+n-(k-1)$ edges is deleted (which does not belong to the $k-1$ edges between Runner and b). After Runner's second move, she has still to cross $k-2$ edges. There are $4k+n-1$ edges left in the game graph; so, Runner does not lose if any of the other $4k+n-1-(k-2)$ edges is deleted. Analogously, after $k-1$ moves of Runner, $k-2$ edges have been deleted already; so, $4k+n-(k-2)$ edges are left in the game graph. If any edge except the one between Runner's current position and b is deleted, Runner will reach b. Generally, in the i-th turn, there are $4k+n-(i-1)$ edges left in the game graph; in order that Runner will still be able to reach b, one of the $3k+n+1$ edges that are not between Runner's current position and b has to be deleted. Altogether,

$$p_{k,n} = \frac{3k+n+1}{4k+n} \cdot \frac{3k+n+1}{4k+n-1} \cdots \frac{3k+n+1}{4k+n-(k-2)} = \prod_{i=0}^{k-2} \frac{3k+n+1}{4k+n-i}.$$

We combine these observations with Lemmas 1.5 and 1.6.

Theorem 1.7. *Given a reachability sabotage game $\mathcal{G}$ on a game graph with n_0 edges, one can construct in polynomial time a randomized reachability sabotage game $\mathcal{G}_{k,n}$ such that for any $k, n \in \mathbb{N}$ with $n \geq n_0$ the following holds: Runner wins the randomized reachability sabotage game $\mathcal{G}_{k,n}$ with a probability $\geq p_{k,n} = \prod_{i=0}^{k-2} \frac{3k+n+1}{4k+n-i}$ iff Runner wins the two-player sabotage game $\mathcal{G}$.*

1.5.3 Towards the PSPACE-Hardness for Arbitrary Probabilities

In order to complete our reduction we have to show that we can adjust the probability $p_{k,n}$ to any ε-neighborhood in $[0, 1]$ and that we can compute the accordant parameters k and n in polynomial time. By taking a closer look at the term $p_{k,n}$ we already see that the probability $p_{k,n}$ can be adjusted arbitrarily close to 0 and arbitrarily close to 1. More precisely, for a fixed $k \geq 2$, we have $\lim_{n\to\infty} p_{k,n} = 1$; and for a fixed n, we have $\lim_{k\to\infty} p_{k,n} = 0$. We will see that the probabilities $p_{k,n}$ form a dense set in the interval $[0, 1]$. Moreover, we will show a stronger result, namely that we can compute efficiently k and n such that n is greater or equal to a given n_0 and the probability $p_{k,n}$ is in an arbitrarily given ε-neighborhood. More precisely, we show the following.

Theorem 1.8. *The set of probabilities $\{p_{k,n} \mid k, n \in \mathbb{N},\ k \geq 2\}$ is dense in the interval $[0, 1]$. Moreover, given $n_0 \in \mathbb{N}$, $p \in [0, 1]$, and $\varepsilon > 0$, there exist $k, n \in \mathbb{N}$ with $k \geq 2$ and $n \geq n_0$ such that $p_{k,n} \in [p - \varepsilon, p + \varepsilon]$; the computation of such k, n, and $p_{k,n}$ is polynomial in the* numerical values *of n_0, $\frac{1}{p}$, and $\frac{1}{\varepsilon}$.*

The proof of this theorem is the subject of Section 1.6; it is rather technical and detached from the game-theoretic arguments in this section.

Note that Theorem 1.8 provides a pseudo-polynomial algorithm since the computation is only polynomial in the *numerical values* of n_0, $\frac{1}{p}$, and $\frac{1}{\varepsilon}$ (and not in their *lengths*, which are logarithmic in the numerical values). For our need – i.e., for a polynomial-time reduction from two-player sabotage games – this is no restriction; the parameter n_0 corresponds to the number of edges in the given two-player game (which has already a polynomial representation), and p and ε are fixed values (i.e., formally they do not belong to the problem instance).

With Theorems 1.7 and 1.8 we can prove our hardness result.

Theorem 1.9. *For each fixed $p \in [0, 1]$ and $\varepsilon > 0$, the following problem is PSPACE-complete: Given a randomized sabotage game $\mathcal{G}'$ with goal set F and a probability $p' \in [p - \varepsilon, p + \varepsilon]$, does Runner win $\mathcal{G}'$ with a probability $\geq p'$?*

Proof. For arbitrary p and $\varepsilon > 0$, we give a reduction from the problem of solving two-player reachability sabotage games to the problem of solving randomized reachability sabotage games where only probabilities in the

interval $[p - \varepsilon, p + \varepsilon]$ are allowed. Note that p and ε do not belong to the problem instance; so they are treated as constants.

Given a two-player reachability sabotage game $\mathcal{G}$, we need to compute a randomized reachability sabotage game $\mathcal{G}_{k,n}$ and a $p_{k,n} \in [p - \varepsilon, p + \varepsilon]$ such that Runner wins $\mathcal{G}_{k,n}$ with a probability $\geq p_{k,n}$ iff Runner wins the two-player game $\mathcal{G}$. Let n_0 be the number of edges of the game graph of $\mathcal{G}$. Then, due to Theorem 1.8 we can compute k and n with $n \geq n_0$ such that $p_{k,n} \in [p - \varepsilon, p + \varepsilon]$. For fixed p and ε, these computations are polynomial in n_0 and hence polynomial in the number of edges in the game graph of the given game $\mathcal{G}$. Since $n \geq n_0$, the result follows immediately from Theorem 1.7, i.e., for the computed k and n we can construct a randomized sabotage game $\mathcal{G}_{k,n}$ in polynomial time such that the following holds: Runner wins the randomized reachability sabotage game $\mathcal{G}_{k,n}$ with a probability $\geq p_{k,n}$ iff Runner wins the two-player sabotage game $\mathcal{G}$. □

As a final remark, we note that we can establish the same refinements of the hardness result as for solving two-player sabotage games: Namely, the problem stays PSPACE-hard if the goal set contains only two vertices and the game graph has only single edges, or the goal set is a singleton set and the graph contains at most double edges (see Theorem 1.4). The former refinement follows immediately since the parametrization gadget in Figure 1.6 contains only two goals and single edges; we have to merge the goal vertices from the other subgraph $\mathcal{G}^n$ with these two goals. For the latter refinement, we must merge the two goals in the parametrization gadget resulting in a parametrization gadget with k double edges to a single goal vertex; then, in the same way we have to merge the goal vertices from the subgraph $\mathcal{G}^n$ with this goal vertex. Merging vertices in the described way does not alter the sum of the edge multiplicities in the graph. Hence, the computation of the probability $p_{k,n}$ is unaffected by these refinements.

1.6 On the Distribution and Computation of the Probabilities $p_{k,n}$

This section deals with the proof of Theorem 1.8: given $n_0 \in \mathbb{N}$, $p \in [0, 1]$, and an $\varepsilon > 0$, we can construct $k \geq 2$ and $n \geq n_0$ in polynomial time with

respect to the numerical values of n_0, $\frac{1}{p}$, and $\frac{1}{\varepsilon}$ such that $p_{k,n}$ is in the interval $[p-\varepsilon, p+\varepsilon]$.

The idea is to first adjust the term $p_{k,n}$ arbitrarily close to 1, and then go with steps of length below any given $\varepsilon > 0$ arbitrarily close to 0; so, we hit every ε-neighborhood in the interval $[0,1]$.

In order to adjust the term $p_{k,n}$ arbitrarily close to 1, we first choose $k = 2$ and a sufficiently high $n \geq n_0$. We will show that it is sufficient to choose $n := \max\left\{n_0, \left\lceil \frac{1}{\varepsilon} \right\rceil\right\}$ (formally, we will also assume $n \geq 4$). For this choice we obtain $p_{2,n} \geq 1-\varepsilon$, which is at least as large as $p-\varepsilon$. Then, we decrease $p_{k,n}$ by stepwise incrementing k by 1 (while keeping n constant). It will turn out that the term $p_{k,n}$ decreases by a value that is lower than $\frac{1}{4k+n+4}$, which is, with the choice of n as above, lower than ε. Iterating this, the values converge to 0, and we hit the interval $[p-\varepsilon, p+\varepsilon]$. Hence, the set $\{p_{k,n} \mid k, n \in \mathbb{N},\ k \geq 2\}$ is dense in the interval $[0,1]$. Furthermore, we will show that it will be sufficient to increment k at most up to $8n$. For this choice, we obtain $p_{k,n} \leq \varepsilon$, which is at least as small as $p+\varepsilon$.

For the complexity analysis, note the following. After each step, the algorithm has to check efficiently whether $p_{k,n} \in [p-\varepsilon, p+\varepsilon]$. The computation of the term $p_{k,n}$ is pseudo-polynomial in k, n, and the test for $p_{k,n} \leq p+\varepsilon$ is in addition polynomial in $\frac{1}{p}$ and $\frac{1}{\varepsilon}$. Since k and n are pseudo-linear in n_0 and $\frac{1}{\varepsilon}$, the whole procedure is pseudo-polynomial in n_0, $\frac{1}{p}$, and $\frac{1}{\varepsilon}$.

Four claims remain to be proven:

- The adjustment of $p_{k,n}$ arbitrarily close to 1 with the proposed choice of n, i.e., given $\varepsilon > 0$, for $n \geq \frac{1}{\varepsilon}$ it holds $p_{2,n} \geq 1-\varepsilon$.
- The adjustment of $p_{k,n}$ arbitrarily close to 0 with the proposed choice of k, i.e., given $\varepsilon > 0$ and $n \in \mathbb{N}$ with $n \geq \frac{1}{\varepsilon}$ and $n \geq 4$, for $k \geq 8n$ it holds $p_{k,n} \leq \varepsilon$.
- The estimation $p_{k,n} - p_{k+1,n} < \frac{1}{4k+n+4}$.
- The test for $p_{k,n} \in [p-\varepsilon, p+\varepsilon]$ is pseudo-polynomial in k, n, $\frac{1}{p}$ and $\frac{1}{\varepsilon}$.

These claims are shown in the rest of this section.

Lemma 1.10. *Given $\varepsilon > 0$, for $n \geq \frac{1}{\varepsilon}$ we have $p_{2,n} \geq 1-\varepsilon$.*

Proof. Since $n \geq \frac{1}{\varepsilon} \geq \frac{1}{\varepsilon} - 8$ for $\varepsilon > 0$, the result follows from

$$p_{2,n} = \frac{n+7}{n+8} \geq 1 - \varepsilon \iff n \geq \frac{1}{\varepsilon} - 8\,.$$

□

Lemma 1.11. *Given $\varepsilon > 0$ and $n \in \mathbb{N}$ with $n \geq \frac{1}{\varepsilon}$ and $n \geq 4$, for $k \geq 8n$ we have $p_{k,n} < \varepsilon$.*

Proof. First note that we have at least $n \geq 1$ and $k \geq 8$. Then

$$\begin{aligned} p_{k,n} &= \prod_{i=0}^{k-2} \frac{3k+n+1}{4k+n-i} \leq \left(\frac{3k+n+1}{3.5k+n}\right)^{\left\lceil \frac{k}{2} \right\rceil} \leq \left(\frac{3k+n+1}{3.5k+n}\right)^{\frac{k}{2}} \\ &\leq \left(\frac{4k+1}{4.5k}\right)^{\frac{k}{2}} \leq \left(\frac{4.125k}{4.5k}\right)^{\frac{k}{2}} = \left(\frac{11}{12}\right)^{\frac{k}{2}} \leq \left(\frac{11}{12}\right)^{4n} < \varepsilon\,. \end{aligned}$$

The inequality $\left(\frac{11}{12}\right)^{4n} < \varepsilon$ remains to be shown. Since $\frac{1}{n} \leq \varepsilon$, it is sufficient to show that $\left(\frac{11}{12}\right)^{4n} < \frac{1}{n}$:

$$\left(\frac{11}{12}\right)^{4n} < \frac{1}{n} \iff n^{\frac{1}{4n}} < \frac{12}{11} \iff \sqrt[n]{n}^{\frac{1}{4}} \leq \sqrt{2}^{\frac{1}{4}} < \frac{12}{11}\,.$$

For $n \in \mathbb{N} \setminus \{0\}$ the inequality $\sqrt[n]{n}^{\frac{1}{4}} \leq \sqrt{2}^{\frac{1}{4}}$ is equivalent to $n^2 \leq 2^n$ and holds for all $n \geq 4$. □

Lemma 1.12. *For $k, n \in \mathbb{N}$ with $k \geq 2$, we have $p_{k,n} - p_{k+1,n} < \frac{1}{4k+n+4}$.*

Proof. In this proof we use the substitution $m := 4k + n + 4$.

$$\begin{aligned} p_{k,n} - p_{k+1,n} &= \prod_{i=0}^{k-2} \frac{3k+n+1}{4k+n-i} - \prod_{i=0}^{k-1} \frac{3k+n+4}{4k+n+4-i} \\ &\leq \prod_{i=0}^{k-2} \frac{3k+n+4+1}{4k+n+4-i} - \prod_{i=0}^{k-1} \frac{3k+n+4}{4k+n+4-i} \\ &= \prod_{i=0}^{k-2} \frac{m-k+1}{m-i} - \prod_{i=0}^{k-1} \frac{m-k}{m-i} \\ &= \prod_{i=0}^{k-1} \frac{m-k+1}{m-i} - \prod_{i=0}^{k-1} \frac{m-k}{m-i} \\ &= \frac{(m-k+1)^{k-1} - (m-k)^{k-1}}{\prod_{i=0}^{k-1} m-i}\,. \end{aligned}$$

Now we use the equation

$$a^l - b^l = (a-b)(a^{l-1} + a^{l-2}b + \cdots + ab^{l-2} + b^{l-1})$$

for the estimation $(d+1)^{k-1} - d^{k-1} = (d+1)^{k-2} + (d+1)^{k-3}d + \cdots + (d+1)d^{k-3} + d^{k-2} \leq (k-1)(d+1)^{k-2}$. We obtain

$$\begin{aligned} p_{k,n} - p_{k+1,n} &\leq \frac{(k-1)(m-k+1)^{k-2}}{\prod_{i=0}^{k-1} m-i} \\ &= \frac{(k-1)(m-k+1)^{k-2}}{m \cdot (m-1) \cdot \prod_{i=2}^{k-1} m-i} \\ &= \frac{k-1}{m(m-1)} \prod_{i=2}^{k-1} \frac{(m-k+1)}{m-i} \leq \frac{k-1}{m(m-1)}. \end{aligned}$$

Since $m > k$ for all $k, n \in \mathbb{N}$, we obtain $p_{k,n} - p_{k+1,n} < \frac{1}{m} = \frac{1}{4k+n+4}$. □

Lemma 1.13. *The computation of the term $p_{k,n}$ is pseudo-polynomial in k and n. The test for $p_{k,n} \leq p + \varepsilon$ is pseudo-polynomial in k and n, and polynomial in $\frac{1}{p}$ and $\frac{1}{\varepsilon}$.*

Proof. First, we rewrite $p_{k,n}$ in the form

$$\frac{(3k+n+1)^{k-1}}{\prod_{i=0}^{k-2} 4k+n-i}.$$

Now, we compute the numerator and the denominator separately. For the computation, we can switch to binary encoding. Each multiplication can be performed in polynomial time in the length of its binary encoding. We need $k-2$ multiplications (for this reason the algorithm is only pseudo-polynomial). The division and comparison of two rational numbers can be done in polynomial time with respect to the length of their binary representations (see von zur Gathen and Gerhard, 2003). So, the quotient $p_{k,n}$ can be computed in pseudo-polynomial time with respect to k and n, and the test to check whether $p_{k,n} \leq p + \varepsilon$ is in addition polynomial in $\frac{1}{p}$ and $\frac{1}{\varepsilon}$. □

1.7 Conclusion and Open Problems

In this chapter we introduced van Benthem's sabotage games. We summarized the result of Löding and Rohde that solving reachability sabotage games is Pspace-complete. We showed that various refined versions of the sabotage game can still be solved in Pspace. In particular this holds for randomized sabotage games and sabotage games with a more general winning condition, e.g., an LTL winning condition or any winning objective for that one can decide in Pspace which player wins a given finite play. Also, we refined the hardness result for randomized sabotage games. We showed that deciding whether Runner wins a game at least with probability p' remains Pspace-complete, even if the p' of the problem instance is restricted to any fixed interval $[p - \varepsilon, p + \varepsilon]$ (with $\varepsilon > 0$).

In our proof of the Pspace-hardness it seems difficult to adjust the probability *exactly* to a given probability p (in our formulation this would mean $\varepsilon = 0$). However, a proof that $p_{k,n}$ cannot be adjusted to a particular rational probability, e.g., $\frac{1}{2}$, is missing. Even if such rational numbers exist, it remains open whether one can refine the construction of the randomized sabotage game for the reduction in Section 1.5 such that Runner's winning probability can be adjusted exactly to any rational probability.

Problem 1.1. Is there any rational number $p' > 0$ such that $p' \neq p_{k,n} = \prod_{i=0}^{k-2} \frac{3k+n+1}{4k+n-i}$ for all k, n? If $p' \neq p_{k,n}$ for all k, n, can one refine the construction of the randomized sabotage game $\mathcal{G}_{k,n}$ in Section 1.5 (possibly using additional parameters) such that Runner's winning probability can be adjusted exactly to any rational number?

We have seen in Section 1.4 that solving two-player reachability sabotage games is only Pspace-hard if the game graph is allowed to contain multi-edges or more than one final vertex. Otherwise they can be solved in linear time. This result, however, is due to the fact that in this case Blocker always removes the edge from Runner's current position to the goal (if such an edge exists). Thus, this proof works for the randomized case only if we ask whether Runner wins with probability 1. Our hardness proof for randomized sabotage games (where probabilities are restricted to a fixed ε-neighborhood) requires at least double edges or more than one final vertex. It remains open whether the construction for the reduction

can be improved to the scenario where the goal set is always a singleton and the game graph contains single edges only.

Problem 1.2. Consider the following problem: Given a randomized sabotage game $\mathcal{G}$ where the game graph is restricted to contain only single edges and only one goal vertex and given a probability p', which is restricted to an a priori fixed interval $[p - \varepsilon, p + \varepsilon]$, does Runner win $\mathcal{G}$ with a probability $\geq p'$? Is this problem PSPACE-hard for any $p \in [0, 1]$ and $\varepsilon > 0$?

Solving randomized reachability sabotage games is closely related to the problem of *dynamic graph reliability* defined by Papadimitriou (1985). There, each edge e fails with probability $p(e, v)$ in each turn, i.e., the probability depends on both the edge e and the current position v of Runner. This allows to adjust the probabilities of the edge deletions after each move; Papadimitriou's proof of the PSPACE-hardness of the dynamic graph reliability problem heavily depends on such adjustments.[7]

For randomized sabotage games, we assume a uniform probability distribution, where in each turn one of the n edges is deleted with probability $p = \frac{1}{n}$. Nevertheless, solving randomized sabotage games is not a special case of the dynamic graph reliability problem since in randomized sabotage games exactly one edge is deleted per turn (as opposed to possibly multiple edge deletions each of which is subject to a given probability). The proof of our hardness result in Section 1.5 depends on this restriction. Sharpening the problem of dynamic graph reliability to probabilities that are independent of Runner's position, one can study the model where in every turn each edge e fails with probability $p(e)$, or even uniformly with probability $\frac{1}{n}$.

Problem 1.3. Consider restrictions of the *dynamic graph reliability* problem (Papadimitriou, 1985) where in every turn each edge e fails with probability $p(e)$, or even each of the n edges fails with probability $\frac{1}{n}$. Are these problems PSPACE-hard?

7 In fact, all problems considered by Papadimitriou (1985, 1994) as *games against nature* allow a precise adjustment of the probability, so that a reduction from the PSPACE-complete problem *SSAT* (Papadimitriou, 1985; Littman, Majercik, and Pitassi, 2001) is possible. Here, SSAT is a stochastic version of QBF (see Papadimitriou, 1994, problem *QSAT*), with randomized quantifiers instead of universal quantifiers.

In Section 1.3 we studied randomized LTL sabotage games, where the sequence of labels of the visited vertices has to fulfill an LTL formula at least with a given probability p. For some applications it might be more convenient to consider branching time logics where the probabilities are incorporated into the logic, e.g., PCTL (Hansson and Jonsson, 1994; also see Baier and Katoen, 2008). We can imagine two different approaches to define semantics of PCTL specifications over sabotage games (also see Klein, 2008, for discussion). The first one is to unfold a randomized sabotage game, which results in a Markov decision process, and then use the standard PCTL model checking (see Baier and Katoen, 2008) on the unfolding. Standard PCTL model checking is polynomial in the size of the Markov decision process. As the unfolding of the sabotage game can be traversed on-the-fly, this approach should result in a PSPACE model checking algorithm.

The second approach to evaluate PCTL specifications is to consider the PCTL formula as a winning condition for Runner. This means that, given a randomized sabotage game and a PCTL formula φ, we ask whether Runner has a strategy such that the resulting probability tree fulfills φ. Clearly, the problem is at least PSPACE-hard, since we can express reachability in PCTL. It is an open issue whether this problem can be solved in PSPACE.

Towards a PSPACE solution for the second approach, a (broken) approach would be to traverse the game tree again on-the-fly in a depth-first manner; then, one could store the probabilities for each subformula in the game tree in a way that only takes the path from the current node to a leaf into account. However, this approach does not work, because it is unclear how to assign probabilities to subformulae for Runner's moves; in general, to decide whether Runner should try to fulfill or violate a subformula, the whole game tree has to be taken into account. Due to this observation, we conjecture that randomized sabotage games with a PCTL winning condition cannot be solved in PSPACE. Then, a task is to find another sufficiently expressive logic with built-in probabilities for which randomized sabotage games can still be solved in PSPACE.

Problem 1.4. Is it possible to solve randomized sabotage games with a PCTL winning condition in PSPACE? Is there another sufficiently expressive logic with built-in probabilities for which this problem is decidable in PSPACE?

Routing Agent vs. Demand Agent

2

Dynamic Network Routing Games

In the face of analyzing routing problems, the games that we present in this chapter are much more involved than the sabotage games of the previous chapter. A sabotage game represents the routing of a single packet only. Also, the sabotage model only allows permanent link failure. In practice, however, not only links could be reestablished after some time, but also various network packets can conflict with each other since the number of packets that can be transmitted between adjacent nodes at the same time is limited.

A *dynamic network routing game* is a two-player game for modeling routing problems in dynamically changing networks that overcomes the aforementioned issues. Compared to the sabotage game we extend the Blocker to a player that we call *demand agent*; he generates packets and blocks certain edges for a certain number of turns (with respect to some constraints). The Runner is generalized to a *routing agent* who can send in each turn one packet per node and outgoing edge to a neighbored node. In this context we also refine the notion of a network. Following the idea that multiple edges between two nodes correspond to different communication

channels, we not only consider multi graphs but also label the edges over a finite index set Σ. Intuitively, the edge labels correspond to the frequency bands that the clients uses to communicate, and the index set Σ represents the set of all available frequencies. The routing agent can use all available channels to transmit packets. Demand agent's actions, however, depend on given constraints; more precisely, they may be subject to the channels that are currently available and to the packets that are currently in the network. We introduce this game model in detail in Section 2.1.

We consider four different winning objectives for routing agent: the delivery of every packet to its destination, the delivery of each packet within a given number of turns ℓ (here also called *ℓ-delivery*), the boundedness of the number of undelivered packets in the network, and the combined form where each packet has to be delivered while the number of undelivered packets in the network has to stay bounded. We provide results on the solvability of routing games, which depend not only on the considered winning objective but also on the form of the given constraints.

In general the constraints may depend on both the blocked channels and the packets in the network. In this case we show general limitations of algorithmic solutions, namely that solving routing games is only decidable for the ℓ-delivery objective but undecidable for the other three objectives (Section 2.2). However, we analyze also simpler versions of the routing game where we restrict the power of the constraints. A natural assumption is that the possibility of a demand (i.e., the possibility to block a channel or to generate a packet) should not be restricted by the packets in the network. This leads to routing games with *weak constraints*, which require that demand agent's possibilities to act depend on the currently blocked edges only. In this case routing games become solvable for the remaining three winning objectives (Section 2.3). However, even under weak constraints a strategy for routing agent to win a given game does in general not result in a suitable routing scheme. Especially, routing agent's decisions are global; they can depend on the entire state of the network (i.e., the packets and the blocked edges in the current network). Therefore, we investigate also routing games with *simple constraints*, in which demand agent has the same possibilities in every turn. More precisely, the possible demands are given by a list of packets that demand agent can generate in every turn (i.e., a list of pairs of source and destination) and two parameters that specify the number of edges that demand agent can block in every turn and the

number of turns for that these edges stay blocked. Under these simple constraints we develop routing algorithms that run locally at each network node. A routing decision at each node u only depends on the packets at u (or, in the case of delivery games, on the packets in a small neighborhood of u) and possibly on the blocked edges in the network (Section 2.4).

The routing game model and parts of the results in this section were developed in discussion and collaboration with James Gross and Wolfgang Thomas. Most of the results on the general model and the model under weak constraints have been published before (see Gross, Radmacher, and Thomas, 2010). The results for simple constraints, which were developed in discussion and collaboration with James Gross and Christof Löding, were not published before.

2.1 The Routing Game Model

A *dynamic network routing game* (or short *routing game*) is played by two players, which we call *demand agent* and *routing agent*. They play on a given multi graph, which represents the network that is the subject of our analysis. The demand agent can generate packets and block edges in the network with respect to given constraints, while the routing agent tries to route the packets to their destination nodes. Formally, a dynamic network routing game is given by a pair

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

where G is the *connectivity graph* and C are the *constraints* for the moves of demand agent. More precisely, a *connectivity graph* G is a labeled multi graph of the form $G = (V, E)$ with finite vertex set V and a finite set E of multi-edges. The set E is partitioned into sets E_a of single edges, where $a \in \Sigma$ for some index set Σ. We write $(u, v)_a \in E$ for the edge (u, v) in E_a. As in the previous chapter, G may be either directed or undirected, but if not stated otherwise, we consider in the following all edges as undirected, i.e., we assume that $(u, v)_a \in E$ implies $(v, u)_a \in E$. However, all results in this chapter hold exactly in the same way for connectivity graphs with directed edges. (Moreover, in Section 2.4, we provide some counterexamples only for graphs with directed edges.) The *constraints* C are a list of rules that describes the conditions imposed on edge removal and packet generation.

In the following we first formalize the notion of packets and blocked edges in the network. Then, we describe the players' moves and the constraints C for demand agent.

Packets, Blocked Links, and Network States

A *packet* consists of a unique identifier from $\mathbb{N}$, its *source*, its *destination*, and a *time stamp*, which is the number of turns since its creation. Thus, a packet is a tuple

$$(id, s, d, k) \in \mathbb{N} \times V^2 \times \mathbb{N},$$

indicating that it has the identifier id, the source node s, the destination node d, and that it was generated k turns before the current point in time. We define the *packet distribution*

$$\lambda\colon V \to 2^{\mathbb{N}\times V^2\times\mathbb{N}}$$

by mapping each node to the set of packets that are currently stored at this node.

The demand agent can block network connections (i.e., edges) for a certain number of turns. The current status of the edges is described by a *blocked-links function*

$$bl\colon E \to \{0, 1, \ldots, m\},$$

which says that a single edge e is blocked for the next $bl(e)$ turns. If $bl(e) = 0$, the edge e is not blocked. Communication between two nodes via the edge e is only possible if $bl(e) = 0$. The maximal number of turns m that can be assigned to an edge for blocking is always given by the constraints C, which we describe later. As E and m are fixed in each routing game, we denote the set of all possible functions bl for a routing game $\mathcal{G}$ by $BL_{\mathcal{G}}$ or simply by BL when the context is clear.

We denote the positions or states of a dynamic network routing game $\mathcal{G}$ as *network states*. Formally, a network state is a pair

$$(\lambda_i, bl_i)$$

consisting of a packet distribution λ_i and a blocked-links function bl_i. We denote by $Q_{\mathcal{G}}$ the set of all network states in the game $\mathcal{G}$. Note that $Q_{\mathcal{G}}$ can be infinite in general since we do not impose an a priori bound on the number of packets in the network.

Moves and Constraints

The dynamics of the packets and blocked links in a routing game arises by the actions of demand agent and routing agent, who make their moves in alternation. We assume here that the demand agent starts (since before demand agent's first move there are no packets in the network that routing agent could deliver). We also assume that no edges are blocked in the beginning of the game. So, the *initial network state* is (λ_1, bl_1) with $\lambda_1(u) = \varnothing$ and $bl_1(e) = 0$ for all $u \in V$, $e \in E$. Stating from the initial network state of a given game $\mathcal{G} = (G, C)$ the players modify the blocked links and the packet distribution that are defined on G. In contrast to the players in a sabotage game, which we discussed in the previous chapter, the two agents in a routing game play ad infinitum. Thus, we define a *play* of the routing game $\mathcal{G}$ as an infinite sequence

$$\pi = \pi_1 \pi_2 \pi_3 \cdots$$

of network states where $\pi_1 = (\lambda_1, bl_1)$ is the initial network state and each step from π_i to π_{i+1} results from the move of demand agent (if i is odd) or routing agent (if i is even).

In detail the two agents modify the packet distribution and the blocked-links function in each move by the following actions. When it is demand agent's move, he generates new packets and blocks edges for a certain number of turns. We restrict demand agent's possible actions in the game $\mathcal{G} = (G, C)$ by a list C of constraints, which we describe in detail shortly. When it is routing agent's move, she can send packets to neighbored nodes. For each node $u \in V$ and each non-blocked, outgoing edge $(u, v)_a \in E$, routing agent can send one packet via this edge.[1] Delivered packets, i.e., packets that reach their destination nodes in this turn, are removed from the network. For all other packets we increment the time stamp by 1. After routing agent's move we decrement the value of the blocked-links function bl by 1 for every edge (if it is not already 0).

1 To model wireless networks it may be more convenient to define routing agent in such a way that she can transmit from each node $v \in V$ at most one packet per frequency $a \in \Sigma$ via some non-blocked edge (as in Gross, Radmacher, and Thomas, 2010). The results for general and weak constraints hold for both models. Also for simple constraints, the proposed routing algorithms can be adapted, but one may run into problems with further extensions of the model, e.g., when interference of frequencies at neighboring nodes is considered.

In the following we describe the constraints C, which specify demand agent's possibilities to move. We start with an example constraint:

$$\text{node } u \text{ has a packet } \wedge \neg \text{ edge } (u,v)_a \text{ is blocked } \longrightarrow \\ \text{block}_2((u,v)_a) \mid \text{block}_1((u,v)_a), \text{generate}(u,v').$$

This constraint says: whenever there exists a packet at node u and the edge $(u,v)_a$ is not blocked, then demand agent can either block the edge $(u,v)_a$ for the next two turns or block the edge $(u,v)_a$ for one turn and generate a packet at node u with destination v'.

All of such constraints in the list C are processed in their given order (in the case that C is a list of more than one constraint). Generally, the *constraints* C are a list of rules of the form

$$\text{Condition} \longrightarrow \text{Behavior}.$$

Here, the *condition* is a Boolean combination of statements of the following form:

1. edge $(u,v)_a$ is blocked
2. node u has a packet (possibly specified in more detail by a source s, a destination d, a time stamp t, or several of these requirements)

The *behavior* is a disjunction (separated by "|") of conjunctions (separated by ",") of demands, i.e., instructions of the following form:

1. $\text{generate}(s,d)$
2. $\text{block}_k((u,v)_a)$

The first instruction says that demand agent can generate a packet at node s with destination d, i.e., he generates the packet $(id,s,d,0)$ at node s where the unique identifier id is set to the lowest available number. The second instruction says that demand agent can block the edge $(u,v)_a$ for the next k turns. Notice that also an edge $(u,v)_a$ with $bl((u,v)_a) \neq 0$ can be blocked again; in this case $bl((u,v)_a)$ is updated to its new value k according to the rule $\text{block}_k((u,v)_a)$.

Per definition the constraints always impose a bound m on the number of turns for that demand agent can block an edge and a bound n on

the number of packets that demand agent can generate per turn. More precisely, we set m to the maximal value of k among all rules of the form $\text{block}_k((u,v)_a)$; if no "block" instruction exists in the constraints, we set $m := 0$. This means that the co-domain of the blocked-links function bl is finite, i.e., the blocked-links function is a function from E to $\{0, 1, \ldots, m\}$ as introduced before. For the bound n on the maximal number of packets generated per turn, we take the overall number of generate instructions contained in the constraints.

The semantics of the constraints is defined in the natural way. In each turn, demand agent processes the constraints in the list in their given order. First, for each constraint demand agent assigns some nodes and edges of the connectivity graph to the nodes and edges that occur in this rule. Then, it is checked whether the condition (left hand side) is true, i.e., it matches the current network state. If this is the case, demand agent can choose at most one of the conjunctions of the behavior (right hand side). In a second step, all statements of the chosen conjunctions are applied on the connectivity graph, i.e., the network state is updated accordingly.

In this chapter, we also consider two restrictions of demand agent's constraints, namely *weak constraints* and *simple constraints*. We call constraints *weak* if all of its conditions (left hand sides) depend on the blocked edges only, i.e., every condition is a Boolean combination of statements that says whether an edge $(u,v)_a$ is blocked.

On a more abstract level, the constraints can be seen as a function $C\colon Q_{\mathcal{G}} \to 2^{Q_{\mathcal{G}}}$, which assigns to each network state of demand agent a set of possible successor network states that are described by the given list of constraints. As weak constraints depend on the blocked links only, weak constraints can be seen as a function $C\colon BL_{\mathcal{G}} \to 2^{Q_{\mathcal{G}}}$, which assigns to each blocked-links function a set of possible successor network states that are described by the given list of constraints.

In a routing game with *simple constraints* demand agent has the same possibilities for packet generation and edge deletion in every turn. Moreover, simple constraints do not describe on which frequencies demand agent blocks edges but only how many edges he blocks. So, demand agent can perform in every turn exactly two actions:

1. generate packets at nodes $s_1, \ldots, s_n$ with destinations $d_1, \ldots, d_n$ and
2. block up to $\#e$ edges, each for the next $\#t$ turns.

A routing game with simple constraints is completely described by a tuple

$$\mathcal{G} = (G, D, \#e, \#t)$$

where G is a connectivity graph, D is an ordered list $((s_1, d_1), \ldots, (s_n, d_n))$ of pairs, and $\#e$ and $\#t$ are two parameters as described above.[2] We denote by $|D|$ the length of the list D. The pairs in the list D can be seen as the *demands* of the network clients, each of which is a packet generated at s_j that should be routed to its destination d_j. The parameters $\#e$ and $\#t$ describe the claims of the environment opposing the network clients. Note that in a routing game with simple constraints at most $\#e \cdot \#t$ edges can be blocked at the same time.

It is easy to formulate simple constraints in the more general framework of general or weak constraints. For this, let the list C of constraints contain the rule

$$\text{true} \longrightarrow \text{generate}(s_1, d_1), \ldots, \text{generate}(s_n, d_n)$$

and $\#e$ times the rule

$$\text{true} \longrightarrow \text{block}_{\#t}(u_1, v_1)_{a_1} \mid \ldots \mid \text{block}_{\#t}(u_k, v_k)_{a_k},$$

where $E = \{(u_1, v_1)_{a_1}, \ldots, (u_k, v_k)_{a_k}\}$ is the set of all edges in the connectivity graph. So, simple constraints are a special case of weak constraints.[3]

Winning conditions

In a dynamic network routing game, the routing agent has to forward the generated packets towards their destinations. More precisely, we distinguish among routing games with four different winning conditions for routing agent:

- For the *boundedness* winning condition we only require that the number of packets in the network is bounded. We say that *routing agent wins a play π of the boundedness game* if there is a k such that in every network state of π the number of packets is $\leq k$.

2 In Section 2.4 we formulate routing algorithms that process the pairs in D in some order. For this reason we define D as a list and not as set or multiset of pairs.

3 Since C consists of $\#e + 1$ rules, the size of C is polynomial in the size of G, D, $\#e$, and $\#t$ only if $\#e$ is given in unary encoding; or we have to encode C in a form that stores the rule for edge blocking only once and its number of repetitions.

- For the *delivery* winning condition we only analyze the packet delivery but neglect that more and more packets may accumulate in the network. We say that *routing agent wins a play π of the delivery game* if in π each generated packet is eventually delivered to its destination.

- The *bounded delivery* winning condition combines the aforementioned winning conditions, i.e., we require both the delivery of all packets and a bound on the number of packets in the network. We say that *routing agent wins a play π of the bounded delivery game* if in π each generated packet is eventually delivered and there is a k such that in every network state of π the number of packets is $\leq k$.

- For the *ℓ-delivery* winning condition we want to guarantee a given delay bound ℓ. We say that *routing agent wins a play π of the ℓ-delivery game* if in π each packet is delivered within ℓ turns after it was generated.

In any of these routing games, demand agent wins a play π if it is not won by routing agent.

Indeed, if routing agent wins a play in a bounded delivery game, she also wins the same play in the delivery and the boundedness game. Also, it is easy to see that each play which routing agent wins in a ℓ-delivery game is a play she also wins for the three other winning conditions.

Remark 2.1. Let π be a play of a routing game. If there exists an ℓ such that routing agent wins π with respect to the ℓ-delivery condition, she also wins π with respect to the delivery, boundedness, and bounded delivery condition.

Proof. Assume that there exists an ℓ such that in π routing agent delivers every packet within ℓ turns. Clearly, in the play π every packet is eventually delivered. Furthermore, the constraints C impose a maximal bound n on the number of packets that can be generated per turn. Since each generated packet is delivered within ℓ turns, there are at most $n \cdot \ell$ packets in the network. Hence, in the play π the number of packets is bounded. □

It is worth pointing out that – even for the bounded delivery condition – the converse does not hold, i.e., there exists a play of a bounded delivery game routing agent wins but in which demand agent wins with

respect to the ℓ-delivery condition for any ℓ. For this, one can imagine a play in which routing agent needs more and more turns to deliver packets while demand agent needs more and more turns to generate these packets.

If we rule out this exotic situation, the boundedness condition is equivalent to the requirement that only finitely many packets have an unbounded delay (delivery time). In the same way the bounded delivery condition then means that every packet is delivered within a bounded number of turns. We will see in a minute that it is always possible to rule out the described situation for bounded delivery games. More precisely, we will show that routing agent can win with respect to the bounded delivery condition if and only if there exists an ℓ such that she can win with respect to the ℓ-delivery condition (Theorem 2.3).

Strategies and Determinacy

A *strategy for demand agent* is a function (here denoted by σ) that maps each play prefix $\pi_1\pi_2\cdots\pi_i$ with an odd i to a network state π_{i+1}; the network state π_{i+1} arises from π_i by a move of demand agent that must be compliant to the given constraints C. In the same way, a *strategy for routing agent* is a function (denoted by τ) that maps each play prefix $\pi_1\pi_2\cdots\pi_i$ with an even i to a network state π_{i+1} that arises from π_i by a move of routing agent. *Demand agent wins the delivery (boundedness, bounded delivery, ℓ-delivery) game* if there exists a strategy σ such that he wins every play π of the delivery (boundedness, bounded delivery, ℓ-delivery) game that is played according to σ. Analogously, *routing agent wins the delivery (boundedness, bounded delivery, ℓ-delivery) game* if she has a strategy τ to win every play π of the accordant routing game. A strategy with which a particular player wins is called a *winning strategy* for this player.

Each of the four routing games defined here is determined, i.e., either demand agent or routing agent has a winning strategy. The determinacy follows from the fact that every game with a Borel type winning condition is determined (Martin, 1975; also see Martin, 1985; Kechris, 1995).

Proposition 2.2. *Dynamic network routing games with a delivery, boundedness, bounded delivery, or ℓ-delivery winning condition are determined.*

Proof sketch. We transform the dynamic network routing game into an *infinite two-player game played on a graph* (see Grädel, Thomas, and Wilke,

2002), in which each vertex corresponds exactly to one network state of the routing game and an indication of which player acts next. Note that it is possible to encode each network state by a natural number since the connectivity graph is finite, the set BL of possible blocked-links functions is finite, and packets are enumerated by natural numbers. Thus, the set of all plays in the unfolded game can be seen as a *Baire space* (see Kechris, 1995). On this topology the determinacy result of Martin (1975) is applicable.

It remains to describe the winning conditions in the form of Borel sets. For the ℓ-delivery condition the unfolded game has the form of a simple safety game that forbids the packets' time stamps to exceed the bound ℓ. The delivery game entails a persistence property for every generated packet, namely that it does not happen that a certain packet never vanishes from some point onwards; hence, the delivery winning condition can be seen as a countable union of these persistence properties. The boundedness condition is countable union of safety properties, each of which says that the number of packets in the network does not exceed a given bound. The winning condition of a bounded delivery game is the intersection of the two aforementioned properties. □

Now, we can clarify the meaning of the bounded delivery condition and its connection to the ℓ-delivery condition. We show that routing agent wins the bounded delivery game if and only if she has a strategy to deliver every packet while guaranteeing bounded delay (delivery time), i.e., there exists some ℓ such that she wins the ℓ-delivery game.

Theorem 2.3. *For routing games with weak constraints, routing agent wins the bounded delivery game iff there exists an ℓ such that routing agent wins the ℓ-delivery game.*[4]

Proof. Assume that routing agent wins the ℓ-delivery game for some ℓ. In this case we already know due to Remark 2.1 that routing agent also wins the bounded delivery game.

Conversely, assume that routing agent wins the bounded delivery game. Thus, routing agent has a winning strategy τ ensuring that every packet is eventually delivered and the number of packets never exceeds some number b. In the following, we first show that, in each play where routing

4 The proof idea for this theorem is due to Christof Löding.

agent moves according to τ, we only have to distinguish between finitely many network states (namely $|Q_{\mathcal{G}}|$ network states). Then, we see that routing agent also has winning strategy, say τ', that needs only finite memory (namely $|S|$ memory sates). Finally, we use these facts to prove that τ' also guarantees that every packet is delivered within a fixed number of turns (namely within $|Q_{\mathcal{G}}| \cdot |S|$ turns).

To show that in each play where routing agent moves according to τ a finite set of network states $Q_{\mathcal{G}}$ is sufficient, we recall that a network state consists of the packet distribution λ and the blocked-links function *bl*. The number of different blocked-links functions is always finite per definition; but the number of different packet distributions could be infinite in general. However, under the assumption that routing agent has a strategy to keep the number of packets bounded by b, the only remaining source of infinity is in the description of the packets (namely the time stamp and the identifier). Since a strategy of routing agent defines the next move based on the complete history of the play, the time stamp can be reconstructed from the history and thus can be omitted without changing the game. Furthermore, we can reuse an identifier one turn after the packet with this specific identifier has been delivered. From a sequence of network states it is then always clear whether a packet was moved or a new packet was created (in the latter case there was no packet with this specific identifier in the network in the previous turn). Since there are at most b packets in the network and a turn consists of two moves (one of demand agent and one of routing agent), at most $2b$ different identifiers are needed when routing agent moves according to τ. Overall, we obtain a finite set of network states $Q_{\mathcal{G}}$.

Towards a finite-memory winning strategy τ' for routing agent, we observe that the boundedness condition – with respect to the bound b – is a simple safety property saying that there are at most b packets in the network. Furthermore, the delivery condition for a packet is a persistence condition, namely that it does not happen that a packet with an identifier *id* never leaves the network. Since we have reduced the number of possibilities for every identifier *id* to $2b$, the delivery condition is a conjunction of $2b$ conditions of the above type. Hence, the winning condition is an intersection between a safety condition and a finite union of persistence conditions. For such kinds of conditions it is known that, if routing agent has a winning strategy (which we assume), she also has a winning strategy,

say τ', that uses only finite memory (see Zielonka, 1998). Let S denote the memory states used by routing agent in τ' to guarantee delivery and boundedness.

We claim that the finite memory strategy τ' also guarantees bounded delivery. Assume that there is a play in which a packet, say P, stays in the network for more than $|S| \cdot |Q_{\mathcal{G}}|$ steps. By the pigeonhole principle, within this period there must be two turns t_1 and t_2 such that the two network states of the game at t_1 and t_2 as well as the two memory states of routing agent's strategy are the same, respectively. Since the strategy of routing agent behaves the same in such situations, demand agent can now simply repeat his moves and thus force a repetition of this loop ad infinitum. As the packet P is not delivered in the loop, it stays in the network forever, contradicting the assumption that routing agent's strategy guarantees the delivery of every packet. We conclude that no packet stays inside the network for more than $|S| \cdot |Q_{\mathcal{G}}|$ turns, if routing agent plays according to τ'. Hence, routing agent wins the ℓ-delivery game for $\ell = |S| \cdot |Q_{\mathcal{G}}|$. □

The Problem of Solving Routing Games

One of our main concerns is to show limitations and possible solutions for routing games with respect to the given constraints. For this purpose we define the routing game problems as the following decision problems.

- *Solving boundedness games:* Given a routing game $\mathcal{G}$, does routing agent win the boundedness game (i.e., does she have a strategy to guarantee that the number of packets in the network stays bounded)?
- *Solving delivery games:* Given a routing game $\mathcal{G}$, does routing agent win the delivery game (i.e., does she have a strategy to eventually deliver any packet)?
- *Solving bounded delivery games:* Given a routing game $\mathcal{G}$, does routing agent win the bounded delivery game (i.e., does she have a strategy to eventually deliver any packet and to guarantee that the number of packets in the network is bounded)?
- *Solving ℓ-delivery games:* Given a routing game $\mathcal{G}$, does routing agent win the ℓ-delivery game (i.e., does she have a strategy to deliver any packet within ℓ turns)?

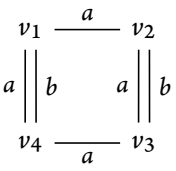

Figure 2.1: The connectivity graph of a dynamic network routing game.

However, we shall also take a broader view on solving routing games. Besides analyzing general limitations and possibilities of routing, our aim is to use winning strategies for routing agent as routing algorithms. We will see that in all cases in which the aforementioned decision problems are decidable we obtain a finite bound on the number of network states that we have to inspect. Thus, our proofs are constructive in the sense that a winning strategy is computable. However, even for routing games with weak constraints the strategies that we obtain are far too complex for our purpose. Our aim is to decompose a winning strategy into several local strategies, each of which runs as a routing algorithm on a single network node (and, if possible, only requires knowledge restricted to a small neighborhood of this node). We tackle this aim in Section 2.4 for routing games with simple constraints. Before we deal with the solvability of routing game problems, let us analyze a tiny example.

Example 2.1. Consider the connectivity graph G in Figure 2.1 with channels over $\Sigma = \{a, b\}$. We define the dynamic network routing game $\mathcal{G} = (G, C)$ where demand agent's constraints C are the following. In each turn, demand agent can generate at node v_1 two packets with destination v_4. He can also block exactly one of the a-labeled edges for one turn; so, at most one of these edges is blocked in every turn. These constraints are weak and can be formalized as follows:

$$\begin{aligned} \text{true} &\longrightarrow \text{generate}(v_1, v_4),\ \text{generate}(v_1, v_4) \\ \text{true} &\longrightarrow \text{block}_1(v_1, v_2)_a \mid \text{block}_1(v_1, v_4)_a \mid \\ &\qquad \text{block}_1(v_2, v_3)_a \mid \text{block}_1(v_3, v_4)_a\,. \end{aligned}$$

First, we analyze the delivery routing game, where the routing agent eventually has to deliver any generated packet. Routing agent wins the

game with the strategy that she sends the packet with the highest time stamp at v_1 to v_4 via the b-labeled edge. This move is always possible, since demand agent cannot block the b-labeled edges in this game. With this strategy, the packet with the highest time stamp always reaches its destination in every turn. So, routing agent wins the delivery game.

Next, we discuss the bounded delivery game. Routing agent does not win with her strategy for the delivery game, because by playing this strategy more and more packets accumulate at v_1. Thus, routing agent has to route packets either via the edge $(v_1, v_4)_a$ or via the path $v_1v_2v_3v_4$. Now, consider that demand agent blocks the edge $(v_1, v_4)_a$. Then, routing agent has to send a packet via the path $v_1v_2v_3v_4$. In this case demand agent can keep this packet at the nodes v_2 and v_3 by deleting the edge $(v_1, v_2)_a$ if the packet is at v_2 and the edge $(v_3, v_4)_a$ if the packet is at v_3. So, this packet will never be delivered. Hence, demand agent wins the bounded delivery game. As our routing games are determined we can apply Remark 2.1 also for the demand agent. It follows that demand agent also wins the ℓ-delivery game for any ℓ.

Surprisingly, routing agent wins the boundedness game. Her strategy is the following. In every turn she delivers one of the generated packets at v_1 directly via the b-labeled edge. She also delivers the other generated packet via the a-labeled edge to v_4 if this edge is not blocked; otherwise she sends this packet to v_2. Furthermore, routing agent sends packets at the node v_2 always to v_3, and she sends packets at v_3 to v_4 whenever this is possible. By playing this strategy, each of the generated packets at v_1 is sent immediately to another node. It is also easy to see that the number of packets at the node v_2 is at most 1 and the number of packets at v_3 is at most 2. Thus, the number of packets in the network is bounded.

2.2 Solvability of Routing Games in the General Game Model

This section deals with the solvability of dynamic network routing games in general, i.e., here we do not impose any restrictions on the constraints. We first show that solving routing games is undecidable for the winning conditions boundedness, delivery, and bounded delivery. Then we show that ℓ-delivery games are solvable, even in this general setting.

Boundedness, Delivery, and Bounded Delivery Games

To show that solving these routing games is undecidable we construct a routing game that simulates a 2-*register machine*. A 2-register machine is a program whose operations modify two registers X_1, X_2. The allowed operations on these registers are the increment and decrement by 1 and the test whether a particular register is 0. Such 2-register machines are Turing complete; in particular both the halting problem, i.e., the question of whether the computation of the register machine eventually stops, and the boundedness problem, i.e., the question of whether the register values stay bounded, are undecidable. In the constructed routing game the number of packets stays bounded if and only if the register values of the 2-register machine stay bounded, and every packet is eventually delivered if and only if the 2-register machine eventually stops.

Theorem 2.4. *Solving dynamic network routing games is undecidable for the boundedness, the delivery, and the bounded delivery winning condition.*

Proof. We reduce the boundedness problem for 2-register machines to the problem of solving boundedness routing games; and we reduce the halting problem for 2-register machines to the problem of solving delivery games and bounded delivery games. For each of these reductions we use the same construction for the routing game. Here, we present 2-register machines in the form

$$\mathcal{R} = I_1; I_2; \ldots; I_k\,.$$

Each I_j with $i \in \{1, \ldots, k-1\}$ is one of the following instructions:

- j: INC X_i, i.e., the increment of the register X_i by 1,
- j: DEC X_i, i.e., the decrement of the register X_i by 1 if $X_i > 0$,
- j: IF $X_i = 0$ GOTO m, i.e., the conditional jump to instruction m,
- j: GOTO m, i.e., the unconditional jump to instruction m.

The last instruction I_k is k: STOP; it stops the computation. Here, we impose some additional assumptions on 2-register machines. We forbid self-loops, i.e., we do not allow that a GOTO instruction I_j points to I_j. In

the same way, we forbid that a GOTO instruction I_j points to the previous instruction I_{j-1} and that two GOTO instructions point directly to each other. This guarantees that at least two other instructions are executed before some instruction is invoked again. Clearly, each 2-register machine can be converted in this format, e.g., one can add an increment instruction followed by a decrement instruction between each pair of consecutive instructions and adapt the GOTO instructions.

We construct, given a 2-register machine $\mathcal{R} = I_1; I_2; \ldots; I_k$, a dynamic network routing game $\mathcal{G} = (G, C)$. The idea is to model each instruction of $\mathcal{R}$ as a vertex in the connectivity graph G. Routing agent sends a packet which represents the program flow of $\mathcal{R}$ through this part of graph. Additional vertices represent the registers and hold as many packets as the indicated register values. Formally, the connectivity graph $G = (V, E)$ has $|k| + 9$ vertices:

$$V = \{v_1, \ldots, v_k, c_1, c_2, c_1', c_2', d_1, d_2, d_1', d_2', t\}.$$

Each of the vertices $v_1, \ldots, v_k$ corresponds to an instruction of the register machine. A packet starting on vertex v_1 with destination v_k will move according to the instructions of $\mathcal{R}$. For technical reasons we also add some additional edges from each vertex v_j (with $j \in \{1, \ldots, k-1\}$) to v_k; demand agent will always be able to block these edges as long as he correctly simulates $\mathcal{R}$. The vertices c_1, c_2 represent the two counters (their values are given by the numbers of packets located at c_1, c_2). In order to decrement a counter the vertex t is used as destination for packets from c_1, c_2. The vertex c_1 is connected to t via the vertex c_1'; analogously, c_2 is connected to t via c_1'. The vertices d_1 and d_1' are adjacent but isolated from the all other nodes; the same holds for d_2 and d_2'. Demand agent uses the edges between these vertices to force routing agent to decrement the number of packets in c_1 and c_2, e.g., if demand agent blocks the edge (d_1, d_1') routing agent has to send a packet from c_1 to c_1'. Due to our additional assumptions on 2-register machines, the construction only uses an edge relation over single edges; it is defined as follows:

$$\begin{aligned} E := &\{(v_j, v_{j+1}) \mid j\text{: IF } X_i = 0 \text{ GOTO } j',\ j\text{: INC } X_i, \text{ or } j\text{: DEC } X_i \in \mathcal{R}\} \\ &\cup \{(v_j, v_{j'}) \mid j\text{: IF } X_i = 0 \text{ GOTO } j' \text{ or } j\text{: GOTO } j' \in \mathcal{R}\} \\ &\cup \{(c_1, c_1'), (c_2, c_2'), (c_1', t), (c_2', t), (d_1, d_1'), (d_2, d_2')\} \\ &\cup \{(v_j, v_k) \mid \text{for all } j \in \{1, \ldots, k-1\}\}. \end{aligned}$$

In the following we give an informal description of the constraints C of the game:

1. Per default the demand agent blocks all edges between the vertices $v_1, \ldots, v_k$ for the next turn; the exceptions are mentioned below. In the same way the demand agent blocks the edges (c_1, c_1') and (c_2, c_2') for the next turn if not stated otherwise. He does not block the edges (d_1, d_1') and (d_2, d_2') unless stated otherwise. The edges (c_1', t) and (c_2', t) are always available. The additional edges from v_j to v_k will always be blocked by demand agent, except for the case that demand agent misbehaves in some way (see below).

2. When there is no packet in the network (especially in the first turn), demand agent creates the packet that mimics the instruction pointer. Formally, he generates the packet $(0, v_1, v_k, 0)$ at vertex v_1. We call this packet the *instruction pointer packet*.

3. If the instruction pointer packet is at vertex v_j, demand agent enables the edge to the vertex v_l that corresponds to the next step in the computation. For example, if the instruction pointer packet is at node u_j and the j-th instruction in $\mathcal{R}$ is IF $X_1 = 0$ GOTO l, then demand agent has to enable the edge (u_j, u_l) if there is a packet at vertex c_1; otherwise he has to enable the edge (u_j, u_{j+1}).

4. Demand agent always blocks the edges that lead to the current instruction for the next *two* turns. This allows demand agent to check in the next turn whether routing agent misbehaves by not sending the instruction pointer packet. More precisely, if the instruction pointer packet is at a node that is incident to a blocked edge, we allow demand agent to create from this turn onwards in every turn a new packet at t with destination v_k, which can never be delivered.

5. If the instruction pointer packet is at vertex v_j and the j-th instruction in $\mathcal{R}$ is INC X_i, demand agent creates a new packet with destination t at vertex c_i. The constraints have to be formulated in such a way that demand agent can only block the edges (v_i, v_k) for all i if demand agent generates the packet at c_i. This prevents demand agent from skipping the generation of the packet at c_i.

6. If the instruction pointer packet is at vertex v_j, the j-th instruction in $\mathcal{R}$ is DEC X_i, and there is a packet at c_i, then demand agent has to enable the edge (c_i, c_i') for one turn. Furthermore demand agent blocks the edge (d_i, d_i') for the next *two* turns. This allows demand agent to check in the next turn whether routing agent misbehaves by not sending a packet from c_i to , c_i'. More precisely, if the edge (d_i, d_i') is blocked and there in no packet at c_i', we allow demand agent to generate from this turn onwards in every turn a packet that can never be delivered. In the same way demand agent can force routing agent to send the packet from c_i' to t: if the edge (d_i, d_i') is not blocked and there is still a packet at c_i', we allow demand agent also to generate packets that can never be delivered. Note that in the case of undirected edges this procedure requires that two decrement instructions are never executed immediately in succession; otherwise routing agent would be able to send a packet from c_i' back to c_i without letting demand agent detect a misbehavior by one the aforementioned checks. Every register machine can be easily modified to fulfill this additional requirement.

7. After the instruction pointer packet arrived its destination v_k, demand agent has to enable the edges (c_1, c_1') and (c_2, c_2') until all packets arrived their destinations.

It is easy to see that demand agent and routing agent will always try to simulate the register machine. Demand agent's constraints are deterministic in the sense that he never has the choice between different actions. He can only skip some actions, which is never an advantage for him. If demand agent skips the generation of a packet at c_1 or c_2, routing agent can deliver the instruction pointer packet immediately. Also, the routing agent has to mimic the instructions of $\mathcal{R}$ as long as the STOP instruction I_k has not been reached. If routing agent misbehaves, the constraints allow demand agent to generate packets form this turn onwards that will never be delivered. As long as $\mathcal{R}$ does not reach the stop instruction I_k, the number of packets at the vertex c_i is equal to the value of the register X_i: demand agent generates a new packet at vertex c_i whenever a INC X_i instruction occurs in $\mathcal{R}$, and routing agent has to deliver a packet at c_i to t whenever the number of packets at c_i is not zero and a DEC X_i instruction occurs in $\mathcal{R}$.

For the reduction to boundedness games, it follows immediately that the number of packet at the vertices c_1, c_2 stays bounded iff register values of $\mathcal{R}$ are bounded. Hence, routing agent wins the boundedness game exactly in this case.

For the reduction to delivery and bounded delivery games, let us assume that $\mathcal{R}$ eventually stops. Then, routing agent eventually delivers the instruction pointer packet. In the following turns, routing agent can deliver all packets while demand agent cannot generate any new packets. So, routing agent wins the delivery and the bounded delivery game. Conversely, if $\mathcal{R}$ never reaches the stop instruction I_k, the instruction pointer packet will never be delivered. So, demand agent wins the delivery and the bounded delivery game in this case. Hence, routing agent wins the delivery as well as the bounded delivery game iff $\mathcal{R}$ eventually stops. □

Note that we can also sharpen the undecidability result. It is still valid if we restrict the set of edge labels Σ to a singleton set, as the constructed connectivity graph only contains single edges. We can also sharpen the result regarding the conditions used in the constraints. In the informally given description of the constraints, we only used statements of the form "edge e blocked" and "node u has a packet". It is not necessary to specify the source, destination, or the time stamp of a packet to obtain undecidability. Hence, our undecidability result does not depend on these features of the routing game model.

Finally, we note that it is also undecidable whether there exists an ℓ such that routing agent wins the ℓ-delivery game. In the presented undecidability proof, routing agent can either deliver all packets within a fixed number of turns (if the register machine stops) or the packet that mimics the instruction pointer can never be delivered (if the register machine never stops). So, we can also apply the reduction for this question. Alternatively, this also follows from the undecidability result for bounded delivery games and Theorem 2.3, which says that deciding whether routing agent can deliver each packet within a bounded number of turns is equivalent to solving bounded delivery games. Thus, solving these two kind of routing games can be reduced to each other.

Remark 2.5. Given a dynamic network routing game, it is undecidable whether there exists an ℓ such that routing agent wins the ℓ-delivery game.

Delivery Games with a Fixed Delay

Solving routing games becomes decidable if demand agent has to deliver every packet within an a priori fixed delay ℓ. These ℓ-delivery games have the form of a safety game where routing agent has to ensure that the time stamp of each packet does not exceed ℓ. Since the number of packets that can be generated in every turn is also limited by a known bound, we only have to inspect a finite state space to solve these safety games.

Theorem 2.6. *For every $\ell \in \mathbb{N}$, solving ℓ-delivery games is decidable.*

Proof. Consider an ℓ-delivery game $\mathcal{G}$. We transform $\mathcal{G}$ into an *infinite two-player game* (see Grädel, Thomas, and Wilke, 2002) on a game graph G' such that each vertex of G' corresponds to a network state of $\mathcal{G}$ and an indication of which player acts next. The edges of G' are directed; they lead from network states where demand agent moves to networks where routing agent moves and vice versa according to the agents' possible actions in $\mathcal{G}$. On this game graph G' routing agent has to win the safety game where she has to avoid any vertex of G' that represents a network state containing a packet with a time stamp $> \ell$. Indeed, solving the ℓ-delivery game $\mathcal{G}$ is equivalent to solving this safety game on the unfolded game graph G'. In the case that G' is finite it is well known that the safety game can be solved efficiently with respect to the size of G' (see Thomas, 1995, 2008a; Grädel, Thomas, and Wilke, 2002).

It remains to be shown that we have to inspect only a finite subset of the network states resulting in a finite game graph G'. Routing agent looses as soon as a packet's time stamp exceeds ℓ. For this reason, when we generate the game graph G', we do not have to generate any successor network state from a network state in which the time stamp of a packet is already $\ell+1$. Also, the number of packets that can be generated in one turn is bounded by the constraints, say by some constant n. So, the vertices in G' only represent network states in which the total number of packets is at most $(\ell+1)\cdot n$. Since each packet gets the lowest available identifier when generated, the identifiers are also bounded by $(\ell+1)\cdot n$. So, in this case a packet distribution λ is a function from V to $2^{[(\ell+1)n]\times V^2\times[\ell+1]}$ where $[k] := \{0,\dots,k\}$. The number of different functions of this form is finite. Since the number of different blocked-links functions is also finite, the size of the game graph G' is finite. □

2.3 Solving Routing Games with Weak Constraints

In this section we show that dynamic network routing games with weak constraints are solvable. The principle idea is to prove bounds on the number of packets in the network that we have to inspect. This yields a solution for the routing game by solving an equivalent safety game that has a finite state space.

We start with general remarks on the possibilities of demand agent's moves under weak constraints (Section 2.3.1). Then, we use these remarks to show that solving boundedness routing games becomes decidable (Section 2.3.2). The main part of this section deals with the solvability of delivery games under weak constraints (Section 2.3.3). Finally, we combine the arguments to solve bounded delivery games (Section 2.3.4).

2.3.1 General Remarks

First of all, let us remember that weak constraints, as defined in Section 2.1, can be seen as a function $C\colon BL \to 2^{Q_G}$, which assigns to each blocked-links function a set of possible successor network states. This leads us directly to the following remarks. The first one says that demand agent cannot gain any advantage by omitting the generation of some packets, because the constraints do not depend on the packet distribution.

Remark 2.7. In a dynamic network routing game with weak constraints, consider a play π that is won by routing agent and results from demand agent playing according to a strategy σ and routing agent playing according to a strategy τ. If demand agent changes his strategy σ to σ' by leaving out the generation of some packets, he will also lose the resulting play, i.e., demand agent cannot improve his strategy in this way.

For the same reason, namely that the constraints do not depend on the packet distribution, the demand agent has the same possibilities to act in all network states that have the same blocked-links function. So, if demand agent can move from a network state (λ_1, bl) to a network state (λ_1', bl'), he can also move from a network state (λ_2, bl) to some network state (λ_2', bl'). Recall that we denote with BL the set of all possible blocked-links function and with $|BL|$ the number of such functions. The second

remark is that demand agent can reach a network work state with a certain blocked-links function within $|BL| - 1$ turns if he can reach such a network state at all.

Remark 2.8. Consider a dynamic network routing game with weak constraints and a network state q_i with blocked-links function bl_i. Let us assume that demand agent has a strategy to reach from q_i a network state q_j with blocked-links function bl_j. Then, demand agent also has a strategy from q_i to reach a network state q'_j with the same blocked-links function, i.e., bl_j, within at most $|BL| - 1$ turns.

Proof. Towards a contradiction assume that demand agent only has a strategy to reach a network state q'_j with blocked-links function bl_j such that he needs from q_i to q_j at least $|BL|$ turns. Then, every resulting play infix $q_i \cdots q'_j$ has at least the length $2|BL|$ (note that for every turn we have one network state for demand agent's move and one network state for routing agent's move). So, there are two network states q_k and q_l with identical blocked-links function where the same player moves (because there are only $|BL|$ different blocked-links functions). We assume that q_k and q_l are network states of demand agent; otherwise we can take q_{k+1} and q_{l+1} since the change of the blocked-links function is deterministic in routing agent's moves. We can optimize demand agent's strategy by choosing in q_k the action that he chooses in q_{l+1}. Repeating this we yield a play infix of a length $< 2|BL|$. Hence, demand agent has a strategy with which he reaches from q_i a network state with blocked-links function bl_j in less than $|BL|$ turns, which is a contradiction to our assumption. □

2.3.2 Boundedness Games

We now show that boundedness games with weak constraints are solvable. The basic idea is to note that due to Remark 2.8 demand agent can loop in some network states that have the same blocked-links function within $|BL|$ turns. We prove an upper bound on the number of packets that routing agent is able deliver in such a loop. Based on this bound we define a threshold for the number of packets in the network. Routing agent tries to keep the number of packets below this threshold. If demand agent has a strategy to exceed this threshold, he can also ensure that the number of packets in the network cannot be bounded at all.

Theorem 2.9. *Given a routing game with weak constraints, routing agent wins the boundedness game iff she can guarantee that there exist at most* $b := |BL| \cdot (n + |E| + 1)$ *packets at each vertex, where* n *is the maximal number of packets that demand agent can generate per turn (given by the constraints) and* $|E|$ *is the number of single edges in the connectivity graph.*

Proof. Clearly, if routing agent guarantees that there always at most b packets in the network, routing agent wins the boundedness game.

For the converse let us assume that demand agent has a strategy to reach a network state with more than b packets in the network. As demand agent's moves do not depend on the packet distribution, we can partition the network states $Q_{\mathcal{G}}$ into a set *Inf* of network states whose blocked-links function can occur infinitely often in a play (i.e., demand agent has a strategy to visit some network states with identical blocked-links function at least twice) and a set *Fin* of network states whose blocked-links function can occur at most once in a play. Since at most $|BL|$ network states in *Fin* are reachable, the sum of packets that can be generated in a play in the states in *Fin* can be bounded by the constant $k := |BL| \cdot n$. From the states in *Inf* demand agent can revisit a network state with the same blocked-links function within $|BL|$ turns (see Remark 2.8). Since demand agent is able to reach a network state that contains more than b packets and at most k of these packets can be generated in network states in *Fin*, demand agent generates at least $b - k = |BL| \cdot (|E| + 1)$ packets in some network states in *Inf*. As demand agent can revisit network states in *Inf* with identical blocked-links function at least every $|BL|$ turns, he also has a strategy to reach a loop of network states with identical blocked-links functions in which he generates at least $|BL| \cdot (|E| + 1)$ packets. Since routing agent can send at most $|E|$ packets to adjacent nodes in each turn, she can deliver at most $|BL| \cdot |E|$ packets in such a loop. So, the number of packets in the network increases with every loop. Hence, demand agent wins the boundedness game. □

With the previous theorem, we can easily reduce the game to a safety game with finite state space where routing agent has to ensure that there are at most b packets at each network node.

Corollary 2.10. *For routing games with weak constraints, solving boundedness games is decidable.*

Proof. The proof is analogous to the proof of Theorem 2.6. We have to solve the safety game on the unfolded game graph where routing agent has to guarantee that the number of packets does not exceed the bound b from Theorem 2.9. Hence, in the unfolding we only have to generate network states with at most $b+1$ packets in the network. As the truncated unfolded game graph is finite, we can solve this safety game efficiently (with respect to the size of the unfolded game graph). □

2.3.3 Delivery Games

Solving delivery and bounded delivery games under weak constraints requires a more involved proof. For the proofs in this section, we use the following terminology. We say that a packet (id, s, d, k) which is currently at vertex u (in a given network state) has the *type* (u, d). So, in a network with a vertex set V the packets have at most $|V|^2$ different types. Further, for a given game, we denote with Δ_{out} the *maximal number of outgoing packets* that routing agent can send per node. In the routing game model considered in this thesis, this is the maximal number of outgoing edges of a node.[5] So, formally we define

$$\Delta_{\text{out}} := \max\{\, |\{(u,v)_a \in E : v \in V, a \in \Sigma\}| : u \in V \,\}.$$

We start with some technical lemmas:

Lemma 2.11. *Given a routing game with weak constraints on a connectivity graph (V, E) with $V = \{v_1, \ldots, v_k\}$. Assume that demand agent can reach a network state with blocked-links function bl where each node v_l stores at least n_{lm} packets with destination v_m (with $l, m \in \{1, \ldots, k\}$). Let $n := \sum_{l,m} n_{lm}$. Then, demand agent can also reach such a state within $|BL|^{n+1} \cdot (\Delta_{out})^n$ turns; moreover it suffices to keep at most $|BL|^n \cdot (\Delta_{out})^{n-1}$ packets of each type for $n > 0$ (and 0 packets for $n = 0$), i.e., all other packets of each type may be discarded after each turn.*

Proof. We show the claim by induction over n. The case $n = 0$ is easy; demand agent has to reach a network state with blocked-links function *bl*,

5 For other network models, where for instance routing agent transmits from each node at most one packet per frequency (as in Gross, Radmacher, and Thomas, 2010), one has to refine this definition.

which he can reach within $|BL| - 1$ turns according to Remark 2.8. For $n > 0$ demand agent has to generate at least n packets, say $P_1, \ldots, P_n$, and thereafter demand agent has to reach a network state q_j with blocked-links function bl such that the packets $P_1, \ldots, P_n$ (or equivalently n other packets of the same types) remain at their vertices where they were generated. We distinguish two cases.

In the first case demand agent has a strategy to generate each packet P_i (with $i \in \{1, \ldots, n\}$), say in a network state q_i with blocked-links function bl_i, without visiting a network state with blocked-links function bl_i twice; and after generating all n packets in this way he reaches a network state q_j with blocked-links function bl where the packets $P_1, \ldots, P_n$ still exist at their required vertices v_{lm}. This is the trivial case where demand agent can reach q_j within $(n+1) \cdot (|BL| - 1) \leq |BL|^{n+1} \cdot (\Delta_{\text{out}})^n$ turns. And since routing agent can send at most Δ_{out} packets from each node in every turn, it suffices for demand agent to keep $(\Delta_{\text{out}} + 1) \cdot (n + 1) \cdot (|BL| - 1)$ packets of each type.

In the second case there exists a packet P_i (in $\{P_1, \ldots, P_n\}$) such that demand agent can reach the network state q_j only by revisiting a network state with some blocked-links function bl_i. We assume due to our induction hypothesis that there is a strategy for demand agent to reach a network state q_j' with blocked-links function bl within $x_{n-1} := |BL|^n \cdot (\Delta_{\text{out}})^{n-1}$ turns where at least the packets $P_1, \ldots, P_{i-1}, P_{i+1}, \ldots, P_n$ exist at their required vertices (and that it suffices to keep at most $x_{n-2} := |BL|^{n-1} \cdot (\Delta_{\text{out}})^{n-2}$ packets of the same type for $n > 1$ and 0 packets otherwise). Now we use that demand agent has a strategy to revisit a network state with blocked-links function bl_i. Demand agent can generate sufficiently many packets of the same type as P_i, so that at least one of these packets remains at its origin after taking the x_{n-1} turns for reaching a network state q_j with blocked-links function bl. Since in the worst case routing agent can send up to Δ_{out} packets per node and turn, demand agent has to visit a network state with function bl_i up to $x_{n-1} \cdot \Delta_{\text{out}}$ times (to accumulate $x_{n-1} \cdot \Delta_{\text{out}}$ packets that have the same type as P_i). For this, demand agent needs at most $x_{n-1} \cdot \Delta_{\text{out}} \cdot (|BL| - 1)$ turns (due to Remark 2.8), and it suffices for him to keep at most x_{n-1} packets of the same type. Then, from this state demand agent needs at most x_{n-1} turns to reach a network state q_j with blocked-links function bl and packets $P_1, \ldots, P_n$ at their required vertices v_{lm}, and it suffices for him to keep at most x_{n-2} packets of the same type (due to

our induction hypothesis). Overall, demand agent can reach q_j within $x_{n-1} \cdot \Delta_{\text{out}} \cdot (|BL| - 1) + x_{n-1} \leq |BL|^{n+1} \cdot (\Delta_{\text{out}})^n$ turns; and it is sufficient for demand agent to keep at most $\max\{x_{n-1}, x_{n-2}\} = |BL|^n \cdot (\Delta_{\text{out}})^{n-1}$ packets of the same type at each vertex. □

Lemma 2.12. *Given a routing game with weak constraints on a connectivity graph (V, E) with $V = \{v_1, \ldots, v_k\}$, and given a network state q with blocked-links function bl where each node v_l stores n_{lm} packets with destination v_m (with $l, m \in \{1, \ldots, k\}$). Assume that demand agent has a strategy from network state q such that one of the mentioned packets can never be delivered. Then, from a network state q' with the same function bl where each vertex v_l stores only $n'_{lm} = \min\{n_{lm}, |BL| \cdot \Delta_{out}\}$ packets with destination v_m in the network, demand agent can also prevent the delivery of a packet.*

Proof. Towards a contradiction we assume that demand agent has a strategy from the network state q to prevent the delivery of at least one packet that is in the network in q, but that routing agent has a strategy τ from the network state q' to deliver all packets that are in the network in q'. If demand agent can prevent the delivery of a packet, he has a strategy to reach a network state q_- with a certain blocked-links function bl_- such that from q_- onwards some particular packet, say P_-, will never be delivered. Due to Remark 2.8 demand agent has a strategy σ to reach such a network state from q within $|BL| - 1$ turns. Also note that routing agent is able to play her strategy τ (which we assumed she has in q') in the network state q. It follows from Remark 2.7 that routing agent can deliver all of the n_{lm} packets of a type with $n_{lm} \leq |BL| \cdot \Delta_{\text{out}}$ and that at least $|BL| \cdot \Delta_{\text{out}}$ of the packets with $n_{lm} > |BL| \cdot \Delta_{\text{out}}$. So, we can assume that the packet P_- has one of the types with $n_{lm} > |BL| \cdot \Delta_{\text{out}}$ in q. Since demand agent has a strategy in network state q to prevent the delivery of a packet, there is a type with $n_{lm} > |BL| \cdot \Delta_{\text{out}}$ such that at least one of the n_{lm} packets of this type will never be delivered (if demand and routing agent play σ and τ from q). Since there are at least $|BL| \cdot \Delta_{\text{out}}$ many packets of this type in q' as well as in q, routing agent can deliver these packets at best in $\frac{|BL| \cdot \Delta_{\text{out}}}{\Delta_{\text{out}}} = |BL|$ turns. So, if the routing agent plays τ in q', there is still at least one of the n_{lm} packets of this type left at v_l after $|BL| - 1$ turns. But according to Remark 2.8 demand agent can reach from q' a network state with blocked-links function bl_- within $|BL| - 1$ turns. So, a packet with

the same type as P_- still resides at its original vertex when demand agent reaches a network state with blocked-links function bl_-. Clearly, by playing σ demand agent prevents the delivery of this packet (because routing agent has the same possibilities to deliver each packet of the same type). This is a contradiction to our assumption that τ is a strategy for routing agent from q' to deliver all packets that are in the network in q'. □

Now we define a variant of the dynamic network routing game where the number of packets in the network is always bounded. For a routing game $\mathcal{G}$, we define the *restricted game* $\mathcal{G}_{\restriction b}$ where we keep at most b packets of the same type and discard all other packets after each move. More precisely, for all vertices u and d, the following happen in $\mathcal{G}_{\restriction b}$ after each player's move: While the number of packets at u with destination d is higher than b, the packet (id, d, t) at u with the highest id is removed from the network. Note that we consider removed packets as delivered; so, the packet deletions in the restricted game are a disadvantage for the demand agent.

Theorem 2.13. *Consider a dynamic network routing game $\mathcal{G}$ with weak constraints, and let $b := (|BL| \cdot \Delta_{out})^{|V|^2 \cdot |BL| \cdot \Delta_{out}}$. Then, routing agent wins the delivery game $\mathcal{G}$ iff she wins the restricted delivery game $\mathcal{G}_{\restriction b}$.*

Proof. Assume that routing agent wins the delivery game $\mathcal{G}$. Towards a contradiction, we assume that demand agent wins the delivery game $\mathcal{G}_{\restriction b}$, say with a strategy σ. We take demand agent's strategy σ for $\mathcal{G}$. Since the constraints C are independent from the packet distribution, routing agent must at least deliver all packets which would not be deleted by the additional rule in the modified game $\mathcal{G}_{\restriction b}$. Since routing agent cannot deliver all packets in $\mathcal{G}_{\restriction b}$, she cannot deliver all packets in $\mathcal{G}$. Hence, demand agent wins $\mathcal{G}$ by playing σ, which is a contradiction to our assumption.

Conversely, assume that routing agent wins the restricted delivery game $\mathcal{G}_{\restriction b}$. Towards a contradiction, we assume that demand agent wins the delivery game $\mathcal{G}$. Then, demand agent has a strategy (for $\mathcal{G}$) to reach a network state q_- where he can guarantee that one of the packets will never be delivered. Due to Lemma 2.12 it suffices to keep at most $|BL| \cdot \Delta_{\text{out}}$ packets of each type from the network state q_- onwards. Since the number of different types is bounded by $|V|^2$, there have to be kept at most $n = |V|^2 \cdot |BL| \cdot \Delta_{\text{out}}$ packets in the network from q_- onwards, so that

demand agent still wins. According to Lemma 2.11 demand agent has a strategy to reach q_- if only $|BL|^n \cdot (\Delta_{\text{out}})^{n-1} \leq b$ packets (and 0 packets if $n = 0$) of each type are kept in the network. So, demand agent also wins the restricted delivery game $\mathcal{G}_{\restriction b}$, which contradicts our assumption. □

So, for a routing game $\mathcal{G}$ with weak constraints, solving the restricted delivery game $\mathcal{G}_{\restriction b}$ with bound b on the number of packets is sufficient for solving the unbounded delivery game $\mathcal{G}$. Although the number of network states of the restricted game is finite, it has not yet the format of a safety game, as ℓ-delivery games (Theorem 2.6) and boundedness games under weak constraints (Theorem 2.9). With the following theorem every restricted delivery game can be turned into an equivalent ℓ-delivery game.

Theorem 2.14. *Given a restricted routing game $\mathcal{G}_{\restriction b}$ with weak constraints, routing agent wins the ℓ-delivery game $\mathcal{G}_{\restriction b}$ for $\ell := |BL|^2 \cdot |V|^3 \cdot b$ iff she wins the delivery game $\mathcal{G}_{\restriction b}$.*

Proof. Clearly, if routing agent wins the ℓ-delivery game, she also wins the delivery game.

For the converse, we assume that routing agent wins the delivery game $\mathcal{G}_{\restriction b}$. We first show that routing agent can send each packet in the network to an adjacent node at least every $|V| \cdot b \cdot (|BL| - 1)$ turns. As $\mathcal{G}_{\restriction b}$ is a restricted routing game, there are at most $|V| \cdot b$ packets at each node (because there exist at most $|V|$ different destinations). For this reason we can assume that routing agent delays a packet P due to another packet with a higher time stamp at most $|V| \cdot b$ times, i.e., she uses at most $|V| \cdot b$ times the available edges for sending other packets instead of using one of the available edges for sending P. Also, we can assume that routing agent has to wait at most $|BL| - 1$ turns until she sends one of the packets of a certain type at a certain node to an adjacent node; otherwise network states with identical blocked-links function would occur at least twice in the meantime (see Remark 2.8). This would imply that demand agent wins by visiting such network states again and again while he can avoid that a packet of a certain type can be sent to an adjacent node. We now give an upper bound on the number of times that a packet visits a certain node before it is delivered. More precisely, we can assume that routing agent sends a packet to the same node at most $|BL| \cdot |V|$ times; otherwise a packet would visit a node twice while also the blocked-links function is

the same (and this would imply that demand agent has a strategy to reach such network states again and again while preventing the delivery of this packet).

Altogether, since routing agent sends each packet to an adjacent node at least every $|V| \cdot b \cdot (|BL| - 1)$ turns, each packet visits each node at most $|BL| \cdot |V|$ times, and the network consists of $|V|$ nodes, routing agent has a strategy to deliver each packet within $|V| \cdot b \cdot (|BL| - 1) \cdot |BL| \cdot |V| \cdot (|V| - 1)$ turns (which is less than $|BL|^2 \cdot |V|^3 \cdot b$ turns). Hence, routing agent wins the ℓ-delivery game $\mathcal{G}_{\uparrow b}$. □

Now we have all ingredients to solve delivery games under weak constraints. We transform the game $\mathcal{G}$ into a restricted game $\mathcal{G}_{\uparrow b}$, for which Theorem 2.13 provides the bound b on the number of packets of the same type at each node. Then, Theorem 2.14 gives us a bound ℓ such that we can solve the restricted game $\mathcal{G}_{\uparrow b}$ with the ℓ-delivery winning condition using Theorem 2.6. We obtain the following result.

Corollary 2.15. *For routing games with weak constraints, solving delivery games is decidable.*

2.3.4 Bounded Delivery Games

We combine some of the previous arguments for boundedness and delivery games to solve bounded delivery games under weak constraints. In particular, we show a variant of Theorem 2.14 where the bound on the number of packets in the network directly follows from the boundedness winning condition provided by Theorem 2.9.

Theorem 2.16. *Given a routing game with weak constraints, routing agent wins the ℓ-delivery game for $\ell := |BL|^2 \cdot |V|^2 \cdot b$ with $b := |BL| \cdot (n + |E| + 1)$ iff she wins the bounded delivery game.*

Proof. Clearly, if routing agent wins the ℓ-delivery game, she also wins the bounded delivery game.

For the converse, we assume that routing agent wins the bounded delivery game. Due to Theorem 2.9 we assume that routing agent guarantees that there are at most $b := |BL| \cdot (n + |E| + 1)$ packets in the network; otherwise demand agent wins the boundedness game and hence also the bounded delivery game. As there are at most b packets in the network,

we can assume that routing agent delays a packet P due to another packet with a higher time stamp at most b times, i.e., she uses at most b times the available edges for sending other packets instead of using one of the available edges for sending P. Also, we can assume that routing agent has to wait at most $|BL| - 1$ turns until she sends one of the packets of a certain type at a certain node to an adjacent node; otherwise some network states with identical blocked-links functions would occur at least twice in the meantime (see Remark 2.8). This would imply that demand agent wins by visiting such network states again and again while he can avoid that a packet of a certain type can be sent to an adjacent node. By a similar argument we can assume that routing agent sends a packet to the same node at most $|BL| \cdot |V|$ times; otherwise a packet would visit a node twice while also the blocked-links function is the same (and this would imply that demand agent has a strategy to reach such network states again and again while preventing the delivery of this packet).

Altogether, since routing agent sends each packet to an adjacent node at least every $b \cdot (|BL| - 1)$ turns, each packet visits each node at most $|BL| \cdot |V|$ times, and the network consists of $|V|$ nodes, routing agent has a strategy to deliver each packet within $b \cdot (|BL| - 1) \cdot |BL| \cdot |V| \cdot (|V| - 1)$ turns (which is less than $|BL|^2 \cdot |V|^2 \cdot b$ turns). Hence, routing agent wins the ℓ-delivery game. □

Again, we can to solve bounded delivery games under weak constraints by solving a safety game. For this, it suffices to solve an ℓ-delivery game with a fixed ℓ, which is provided by the previous theorem. Additionally, we may also bound the number of packets at each node according to Theorem 2.9 in order to reduce the state space that we have to explore.

Corollary 2.17. *For routing games with weak constraints, solving bounded delivery games is decidable.*

Finally, we can also decide for routing games with weak constraints whether there exists an ℓ for that routing agent wins the ℓ-delivery game. This follows immediately from the previous result, because the question is equivalent to solving bounded delivery games (Theorem 2.3)

Remark 2.18. Given a dynamic network routing game with weak constraints, one can decide whether there exists some ℓ such that routing agent wins the ℓ-delivery game.

2.4 Solving Routing Games with Simple Constraints

This section deals with solving dynamic network routing games under simple constraints. Besides determining the winner of a given routing game, the aim of this section is to compute an efficient routing algorithm. Whenever routing agent wins a given routing game, we want to synthesize a winning strategy that has a simple format and allows fast routing decisions at each network node. More precisely, a routing algorithm should

- run locally at each network node u, i.e., in each turn the routing decisions at u are independent from the routing decisions at other network nodes,
- provide fast routing decisions for the packets at u after each update of the network state by demand agent (in the form of generated packets and blocked edges), and
- provide routing decisions only depending on the packets at u (or, in the case of delivery games, only depending on the packets in a small neighborhood of u). Also, if possible, the algorithm should only depend on the blocked edges in a small neighborhood of u.

In this section we provide routing algorithms for delivery, boundedness, and bounded delivery games with simple constraints.

For delivery games we will see that, whenever routing agent wins, she has a winning strategy that only depends on a local neighborhood of each node. In this way we obtain a routing algorithm, for which routing decisions at each node u only depend on the packets at u and on the blocked edges in the $\#t$-hop neighborhood of u. Also the current turn number is used to associate a fixed time slot with each packet, in which only this single packet is delivered. For the case of $\#t = 1$ we obtain a more efficient algorithm, which routes packets simultaneously. In this case, the routing decisions only depend on the packets at u and the available incident edges (Section 2.4.1).

For boundedness games we will present a simple routing algorithm, which associates with each packet a distinct path when this packet is generated. The algorithm provides a solution for all scenarios where – as long

as the blocked edges do not change – fixed forwarding paths are sufficient. Using this algorithm the routing decisions at a node u only depend on the blocked edges in the network and the packets at u (Section 2.4.2).

Finally, we will discuss bounded delivery games, for which we refine the routing algorithm that we used for boundedness games. The refined algorithm only works under additional assumptions on both the demands and the paths for the packets. More precisely, we need an additional fairness assumption saying that a connection between each pair of adjacent nodes is available again and again; and the algorithm requires that the paths for routing the packets cannot be combined to any loop (Section 2.4.3).

2.4.1 Delivery Games

First, we develop a local routing algorithm for delivery games with simple constraints. The central observation is that it is sufficient for routing agent to route only one packet at the same time to its destination. This leads us to the idea to extend the notion of sabotage games, which we discussed in Chapter 1. More precisely, we know that demand agent wins a delivery game if and only if he has a strategy to generate a packet that will never be delivered. Hence, it suffices to check for every pair (s, d) of source and destination node separately whether demand agent can prevent the packet delivery. We introduce so-called *extended sabotage games* where the player Runner starting at vertex s has to reach vertex d. Blocker's actions of edge removal are adapted to demand agent's actions in routing games with simple constraints; also, edges are restored as in routing games. We show that solving a delivery game $\mathcal{G}$ under simple constraints is equivalent to solving the extended sabotage game for all pairs of source and destination provided with $\mathcal{G}$. Based on this result we derive a local routing algorithm for solving delivery games under simple constraints. Finally, we discuss the special case of $\#t = 1$, i.e., the case that demand agent completely determines the set of blocked edges in every turn (as opposed to determining only a subset of the blocked edges in every turn).

Extended Sabotage Games

An *extended sabotage game* is a tuple

$$\mathcal{S} = (G, s, d, \#e, \#t)$$

where $G = (V, E)$ is a connectivity graph,[6] $s \in V$ is the initial vertex where Runner starts, $d \in V$ is the goal vertex which Runner has to reach, $\#e$ is the number of edges which Blocker may block per turn, and $\#t$ is the number of turns for which an edge stays blocked (if not blocked again in the meantime).

Blocker blocks edges in the extended sabotage game according to the parameters $\#e$ and $\#t$ in the same way as in the dynamic network routing game with simple constraints. So, there are at most $\#e \cdot \#t$ edges blocked in the network at the same time. A position of the extended sabotage game is a tuple $(v, E_1, \ldots, E_{\#t})$, where v is the vertex where Runner currently resides and each E_i is a set containing at most $\#e$ single edges; these are the edges that are blocked for the next i turns. Hence, the sets E_i are pairwise disjoint. In contrast to usual sabotage games, the initial position of an extended sabotage game depends on *initial edge blockings* of Blocker. More precisely, at the beginning of the game Blocker has the opportunity to block $\#t$ times $\#e$ edges. The resulting initial position is $(s, E_1, \ldots, E_{\#t})$, where Runner resides at the initial vertex s and each set E_i contains at most $\#e$ edges (and all sets E_i are pairwise disjoint). Then, starting from this initial position Runner and Blocker move in alternation as in a usual sabotage game. From a position $(u, E_1, \ldots, E_{\#t})$, first Runner traverses the graph from the vertex u via a non-blocked edge (u, v), i.e., she chooses an edge from the set $(\{u\} \times V) \setminus (E_1 \cup \ldots \cup E_{\#t})$. The position is updated to $(v, E_1, \ldots, E_{\#t})$. Then, Blocker blocks $\#e$ edges, say the edges of a set $B = \{e_1, \ldots, e_{\#e}\} \subseteq E$. The edges in E_1, which were previously blocked for the next turn only, become available again if they are not in B. The new position is $(v, E_2 \setminus B, \ldots, E_{\#t} \setminus B, B)$.

A *play* is an infinite sequence of positions

$$\pi = (u_0, E_1^0, \ldots, E_{\#t}^0)(u_1, E_1^0, \ldots, E_{\#t}^0)(u_1, E_1^1, \ldots, E_{\#t}^1)\cdots$$

where $(u_0, E_1^0, \ldots, E_{\#t}^0)$ is the initial position (after Blocker's initial edge blockings) and the subsequent positions arise by moves of Runner and Blocker as described above. *Runner wins a play π of the extended sabotage*

6 As in routing games with simple constraints we do not distinguish between different frequencies in extended sabotage games. Hence, a connectivity graph can be seen in this context as a multi graph without edge labels. We use here the concept of connectivity graphs as we borrow the graph directly from the dynamic network routing game.

game $\mathcal{S}$ if she reaches the goal vertex. In contrast to the sabotage games discussed in Chapter 1, plays are infinite as edges are also restored after some time (except for the case that $\#e \cdot \#t \geq |E|$). However, a play can be considered as finite if Runner reaches the goal vertex. Plays that are won by Blocker are infinite.

A strategy for Runner is a function from $(V \times (2^E)^{\#t})^+$ to V which maps each play prefix to the vertex to which Runner moves next or to Runner's current vertex if she skips. A strategy for Blocker is given by some initial edge blockings and a function which maps each play prefix to a set B containing the $\#e$ edges that Blocker blocks next. *Runner wins an extended sabotage game* $\mathcal{S}$ if she has a strategy to win every play of $\mathcal{S}$ in which she moves according to this strategy. In the same way *Blocker wins* $\mathcal{S}$ if he has a strategy to win every play in which he blocks edges according to this strategy. It is easy to see that extended sabotage games are positional determined, as discussed for the original sabotage games in Chapter 1, i.e., for each extended sabotage game exactly one of the players has a positional winning strategy.

Proposition 2.19. *Extended sabotage games are determined. Moreover, the winning player always has a positional winning strategy.*

This implies that extended sabotage games can also be evaluated after a finite number of turns. When Runner wins with a positional strategy, each position occurs at most once until she reaches a final vertex. So, we can evaluate a play as winning for Blocker if a position occurs twice before Runner visits a final vertex. Indeed, one of the two cases always occurs since the number of positions is finite.

Properties of Runner's Winning Strategies in Extended Sabotage Games

To use Runner's winning strategies for routing in delivery games with simple constraints, we analyze two further aspects. First, towards a local routing algorithm we show that Runner only needs to know about the blocked edges in the $\#t$-hop neighborhood of her current vertex. Second, we show that Runner can reach a final vertex in a certain number of turns (if she wins at all). Since we want to route the packet in the delivery game one after the other, this bound can be used as an upper bound on the number of turns after which we can begin to transmit the next packet.

Towards a local strategy for Runner, we observe that it cannot be useful for Blocker to block an edge which will be restored at least as soon as Runner reaches an incident vertex. To formalize this, we define the *k-hop-neighborhood* $N_k(u)$ of a node u as the set of edges which occurs in any path of length at most k starting at u (where the length of a path corresponds to its number of edges), i.e.,

$$N_k(u) = \{(v_{i-1}, v_i)_a \in E \mid i \in \{1, \dots, l\} \text{ and there is a path } v_0 v_1 \cdots v_l \text{ with } v_0 = u \text{ and } l \leq k\}.$$

Because each edge is restored $\#t$ turns after blocking (if Blocker does not block it again), neither of the players has to care about the blocked edges outside the $\#t$-hop neighborhood of Runner's current vertex u. If an edge outside the $\#t$-hop neighborhood of u is blocked, it will be available again unless Blocker chooses this edge for blocking in a future turn.

Proposition 2.20. *In each extended sabotage game $\mathcal{S} = (G, s, d, \#e, \#t)$ where Runner (Blocker) wins, she (he) also has a winning strategy which only depends on Runner's current vertex and on the edges that are blocked in the $\#t$-hop neighborhood of Runner's current position.*

Now we show an upper bound on the number of turns that Runner needs to reach the goal vertex (in the case that she has a winning strategy). For this, we give an upper bound on the number of times that Runner has to revisit a node.

Lemma 2.21. *If Runner wins an extended sabotage game $\mathcal{S} = (G, s, d, \#e, \#t)$, Runner also has a winning strategy with which she visits each vertex $v \in V$ at most $(|N_{\#t}(v)| + 1)^{\#e \cdot \#t}$ times.*

Proof. Note that sabotage games are positional determined; and for both players strategies suffice that only depend on the $\#t$-hop-neighborhood of Runner's current position. Also note that in an extended sabotage game there are at most $\#e \cdot \#t$ edges blocked in the graph. For each vertex $v \in V$ we can overapproximate the number of combinations of blocked edges in $N_{\#t}(v)$ by counting the number of different functions from $\{1, \dots, \#t\}$ to $(N_{\#t}(v) \cup \{\perp\})^{\#e}$; such function assigns to each number $i \in \{1, \dots, \#t\}$ the set of edges which are blocked for the next i turns ($\perp$ is used as a placeholder if less than $\#e$ edges are blocked for i turns). There exist

$(|N_{\#t}(v)| + 1)^{\#e \cdot \#t}$ such functions. So, at the latest when Runner visits v more than $(|N_{\#t}(v)| + 1)^{\#e \cdot \#t}$ times, a position is repeated where Runner is at v and exactly the same edges are blocked in the $\#t$-hop neighborhood $N_{\#t}(v)$. Hence, if Runner wins $\mathcal{S}$, she has a winning strategy with which she visits each vertex v at most $(|N_{\#t}(v)| + 1)^{\#e \cdot \#t}$ times. □

For a sabotage game $\mathcal{S} = (G, s, d, \#e, \#t)$ with $G = (V, E)$ we define

$$bound(\mathcal{S}) := \sum_{v \in V} (|N_{\#t}(v)| + 1)^{\#e \cdot \#t}.$$

This serves as an upper bound on the number turns that Runner needs to win an extended sabotage game (if she wins at all). Note that the bound $bound(\mathcal{S})$ does not depend on the initial vertex s and the goal vertex d.

Corollary 2.22. *If Runner wins an extended sabotage game $\mathcal{S}$, Runner has a winning strategy to win $\mathcal{S}$ within $bound(\mathcal{S})$ turns.*

We recall that in the original two-player sabotage game, which we discussed in Chapter 1, Runner has to visit each vertex at most once. Moreover, in the two-player as well as in the randomized version of the sabotage game the length of the play is bounded by the number of edges. We used this polynomial bound as a key to prove that solving sabotage games is Pspace-complete. For extended sabotage games, however, it is an open problem whether a play can be evaluated as winning for either of the two players after polynomially many turns.

In the case that $\#e = 1$ and the parameter $\#t$ is equal to the number of edges, we obtain a usual sabotage game, for which we know that solving them is Pspace-hard. Hence, solving extended sabotage games is also at least Pspace-hard. But since we only have an exponential bound on the number of turns that Runner needs to win, we cannot borrow the Pspace solution method known from the original sabotage games. However, it is easy to see that solving extended sabotage games is in ExpTime, since the unfolding of a given extended sabotage game games yields an *infinite two-player game* (see Thomas, 2008a; Grädel, Thomas, and Wilke, 2002) on a game graph which size is exponential in the given game. On the unfolding one can solve the reachability game in linear time. However, it is an open issue to find a tight complexity bound for solving extended sabotage games.

From Extended Sabotage Games to Delivery Routing Games

We show that the problem of solving delivery games under simple constraints can be reduced to solving extended sabotage games.

Theorem 2.23. *Given a dynamic network routing game $\mathcal{G} = (G, D, \#e, \#t)$ with simple constraints and $D = ((s_1, d_1), \ldots, (s_n, d_n))$, we define for each pair of source and destination the extended sabotage game $\mathcal{S}_j = (G, s_j, d_j, \#e, \#t)$. Then, routing agent wins the delivery game $\mathcal{G}$ iff Runner wins each of the extended sabotage games $\mathcal{S}_1, \ldots, \mathcal{S}_n$.*

Proof. Assume that Runner wins the extended sabotage game $\mathcal{S}_j$ for every $j \in \{1, \ldots, n\}$. We claim that routing agent can deliver all generated packets in $\mathcal{G}$ by routing only one packet at once. More precisely, routing agent chooses one of the packets with the highest time stamp and route it according to Runner's sabotage-game strategy. Before routing agent begins to transmit such a packet (id, s_j, d_j, t), the packet is at its source node s_j and some sets of edges are blocked, say $E_1, \ldots, E_{\#t}$, where each E_j contains the edges that are blocked for the next j turns. Routing agent sends the packet according to Runner's strategy in the extended sabotage game $\mathcal{S}_j$ where the initial position is $(s_j, E_1, \ldots, E_{\#t})$ after Blocker's initial edge blockings. Then, the actions of demand agent and Blocker as well as the actions of routing agent and Runner coincide until the packet is delivered. Note that routing agent does not send any other packet as long as this single packet has not been delivered. Thus, after she delivers this packet, all other packets are still at their respective source nodes. Then, routing agent delivers the next packet according to the sabotage-game strategy. As Runner wins each of the extended sabotage games, routing agent eventually delivers every generated packet. Hence, she wins the delivery game $\mathcal{G}$.

Conversely, assume that there exists a $j \in \{1, \ldots, n\}$ for which Blocker wins the sabotage game $\mathcal{S}_j$. Then, demand agent can use Blocker's strategy for $\mathcal{S}_j$ to prevent in $\mathcal{G}$ the delivery of one of the packets generated at s_j. For this, demand agent starts blocking the $\#e \cdot \#t$ edges according to Blocker's initial edge blockings in $\mathcal{G}_j$. Also, in the same turn as in which Blocker performs the last of these initial edge blockings, demand agent generates one packet at s_j with destination d_j. Then, demand agent always blocks edges according to Blocker's sabotage-game strategy for $\mathcal{S}_j$ where Runner's

current vertex corresponds to the position of the generated packet in $\mathcal{G}$. Since Blocker wins $\mathcal{S}_j$, demand agent prevents the delivery of the generated packet in $\mathcal{G}$. Hence, he wins the delivery game $\mathcal{G}$. □

The proof of the previous theorem is constructive in the sense that it provides a positional strategy for both routing agent and demand agent. In contrast to Runner's strategy in the extended sabotage games $\mathcal{S}_j$, however, the constructed strategy for routing agent lacks the property to depend on the $\#t$-hop neighborhood only. In order to know whether routing agent can begin to route a packet, she has to check whether any previously transmitted packet has reached its destination. This means that in general routing agent has to be aware of the packets at all network nodes. She also has to agree on one of the packets with the highest time stamp.

To avoid these problems we partition the turns into time slots, each of which suffices to deliver any packet to its destination. We set the length of the time slots to the bound on the number of turns that Runner needs to win an extended sabotage game, which is provided by Corollary 2.22. Again, we route only one packet at a time; but instead of checking whether any previously transmitted packet has reached its destination, we route packets from different source nodes in a round-robin fashion and grand routing agent $bound(\mathcal{S}_j)$ turns to deliver each packet.

This idea is implemented in Algorithm 2.1. There, we define exactly one time slot for all pairs in $((s_1, d_1), \ldots, (s_n, d_n))$ which have the same source node s_j. Here, we assume that we have $k \leq n$ such time slots, which corresponds to the number of nodes where packets are generated.

Whenever the time slot for the node s_j begins, always the packet with the highest time stamp at s_j is chosen. In the remaining turns of this time slot, only this packet is routed at s_j (in the case that the packet revisits s_j). In the turns which do not belong to the time slot of s_j, we only route packets whose source node differs from s_j. So, in each time slot for some node s_j exactly one packet, whose source is s_j, is routed to its destination.

The following precomputations are needed to run the algorithm. The maximal time that is needed to deliver a packet is provided by the parameter *slot-time*; we set this parameter to $bound(\mathcal{S}_j)$, which is the same value for every $j \in \{1, \ldots, n\}$. The time for all slots is given by the parameter *round-time*; since we have k nodes where packets are generated, we set this parameter to $k \cdot$ *slot-time*. The parameter *offset* is the time offset of

Algorithm 2.1: A routing algorithm that guarantees packet delivery under simple constraints.

Input: routing game $\mathcal{G}$ with simple constraints, own node name u, *slot-time*, *round-time*, *offset*, winning strategies $\tau_1, \ldots, \tau_n$ for the extended sabotage games $\mathcal{S}_1, \ldots, \mathcal{S}_n$

```
1  while true do (* repeat forever *)
2    get global-timestamp
3    if offset ≤ global-timestamp mod round-time < offset + slot-time
     then (* own time-slot – only route packets with source u *)
4      if offset = global-timestamp mod round-time then
         (* own time slot just started – choose a new packet to route *)
5        if there exists a packet at current node u then
           (* when a time slot starts, every packet at u has source u *)
6          choose the packet (id, s_j, d_j, t) that has the highest
           time stamp t
7          current-packet := id
8          route packet (id, s_j, d_j, t) according to its sabotage-game
           strategy τ_j for the game S_j
9        end if
10     else (* own time-slot continues – only route the current packet *)
11       if there exists a packet (id, s_j, d_j, t) with id = current-packet
         then
12         route packet (id, s_j, d_j, t) according to its sabotage-game
           strategy τ_j for the game S_j
13       end if
14     end if
15   else (* foreign time slot – only route packets with a source that differs
     from u *)
16     if there exists a packet (id, s_j, d_j, t) with s_j ≠ u then
17       route packet (id, s_j, d_j, t) according to its sabotage-game
         strategy τ_j for the game S_j
18     end if
19   end if
20   wait for the next turn
21 end while
```

each source node; for the j-th source node ($j \in \{1, \ldots, k\}$), we set the offset to $(j-1) \cdot \textit{slot-time}$. Indeed, since the algorithm runs on each node independently, a global time stamp is needed to make use of the time frames in a round-robin fashion; we assume here that the algorithm can obtain this information via a "get *global-timestamp*" command.

Compared to the strategy in the proof of Theorem 2.23 the local strategy from Algorithm 2.1 has the disadvantage that it leads to higher packet delivery times since routing agent always waits for the next time slot before she routes the next packet. However, the proposed algorithm leaves much room for improvements. On the one hand the bound $\textit{bound}(\mathcal{S}_j)$ provided by Corollary 2.22 could be improved. On the other hand packets can be routed simultaneously as long as a bottleneck of two or more packets does not lead to a situation where demand agent can cut off a packet from reaching its destination. It's a challenging problem to detect these situations in a refined analysis of the network structure and the sabotage-game strategies.

An Improvement for $\#t = 1$

Finally, we discuss the special case of $\#t = 1$, i.e., the case that demand agent completely determines the set of blocked edges in every turn. Indeed, we could also obtain a solution for this case by solving the extended sabotage game with $\#t = 1$ for each pair of source and destination. However, we discuss the solution separately, since in this case the extended sabotage games, which we used to solve routing games, have a very simple form.

More precisely, the number of edges that Blocker can delete in his initial edge blocking is equal to the number of edges that Blocker can delete after each of Runner's moves. As a consequence Blocker's ability to block a certain set of edges does not depend on his previous moves anymore. So, in each move Blocker chooses $\#e$ edges, and these are exactly the edges that are blocked for Runner's next move. Clearly, in an extended sabotage game with $\#t = 1$ it suffices for Runner to visit each vertex at most once. If Runner visits a vertex twice, Blocker will ensure that Runner can move exactly to the same positions as she visited this vertex for the first time. This means that Runner wins an extended sabotage game $\mathcal{S} = (G, s, d, \#e, 1)$ if and only if she wins $\mathcal{S}$ within $|V|$ turns. Moreover, Runner wins $\mathcal{S}$ if and only if she can move from s to an adjacent node s' and wins the extended

sabotage game $\mathcal{S}' = (G, s', d, \#e, 1)$ within $|V| - 1$ turns from s'. She then proceeds in a similar fashion.

In an equivalent formulation this means that Runner wins an extended sabotage game $\mathcal{S} = (G, s, d, \#e, 1)$ if and only if the following holds: There exists a path of length at most $|V|$ from s to d for each edge blocking; from the next node on each of these paths there exists a path of length $|V| - 1$ to d for every edge blocking, and so on. This enables us to compute strategies for such a game inductively in an efficient way. Given a game $\mathcal{S} = (G, s, d, \#e, 1)$ we first mark the goal vertex d as a vertex from which Runner wins within 0 turns. Then, for each $i \in \{1, \ldots, |V|\}$ we mark each vertex u as a vertex from which Runner wins within i turns if – against all possible edge removals of Blocker – there exists an edge from u to a vertex from which Runner wins within $i - 1$ turns. This is the case if there exist at least $\#e + 1$ edges from u to vertices from which Runner wins within $i - 1$ turns. We can compute the sets of vertices from which Runner wins within i turns iteratively. The computation can be seen as an "attractor" construction for a reachability game (see for instance Thomas, 2008a, section 4.1). So, we can check whether Runner wins $\mathcal{S}$ in time $O(|V| + |E|)$ for a connectivity graph $G = (V, E)$, and if this is the case, we also obtain a winning strategy in this time.

Transferred to delivery routing games with $\#t = 1$ this has two consequences. First, we can compute a winning strategy for routing agent in time $O(|D| \cdot (|V| + |E|))$ if she wins the delivery game. Second, packets cannot be cut off from their destination as long as they are only routed according to their sabotage-game strategy or are not routed at all. This means that routing agent can try to route all packets in the network simultaneously. To guarantee that no packet is delayed infinitely long, it suffices that routing agent tries to transmit packets with a higher time stamp first. We implemented this idea in Algorithm 2.2.

2.4.2 Boundedness Games

We now develop a simple routing algorithm for boundedness games with simple constraints. To obtain a convenient and feasible solution, we assume that our routing algorithm has to fulfill an additional requirement, namely that generated packets for each pair of source and destination are forwarded via the same path as long as the same set B of edges is blocked. This means

Algorithm 2.2: A routing algorithm that guarantees packet delivery under simple constraints if $\#t = 1$.

Input: routing game $\mathcal{G}$ with simple constraints and $\#t = 1$, own node name u, strategies $\tau_1, \ldots, \tau_n$ for the extended sabotage games $\mathcal{S}_1, \ldots, \mathcal{S}_n$

```
1 while true do (* repeat forever *)
2   mark all packets at u
3   while there exists an marked packet at u do
4     choose the packet (id, s_j, d_j, t) with the highest time stamp t
      and unmark it
5     route the packet (id, s_j, d_j, t) according to its sabotage-game
      strategy τ_j for the game S_j if an accordant edge is still available
      for transmission
6   end while
7   wait for the next turn
8 end while
```

that – as long as B stays fixed – the packets generated due to a certain pair always take the same path. However, when the set of blocked edges changes, this property may be violated for several turns.

Under this assumption, the major insight is that every pair of source and destination in D needs its own independent path, i.e., a path which does not share an edge with a path connecting another pair in D.[7] As demand agent can block up to $\#e \cdot \#t$ edges, we have to find independent paths in every graph which results from the given connectivity graph by removing up to $\#e \cdot \#t$ edges. So, we need to map each set B of blocked edges (with $|B| \leq \#e \cdot \#t$) to the independent paths from the source to the destination nodes. In the following we refer to this mapping as a *scheme*. Whenever the edges in the set B are blocked, routing agent should route the packets via the independent paths which the scheme associates to B. Later, we will see that the existence of a scheme exactly characterizes the boundedness games where routing agent can win with a routing algorithm

7 Note that we refer here with "edge" to a particular labeled edge from a nodes u to a node v in the connectivity graph. So, another path may also contain an edge from u to v if this edge has a different label.

that fulfills our additional assumption on the forwarding paths. However, we will also see that schemes some boundedness games can only be solved without our additional requirement. So, schemes do not provide a solution to every solvable boundedness games.

Schemes

We first formally define a *list of independent paths*. For a connectivity graph $G = (V, E)$ with edge labels over an index set Σ, we call a list of paths $(v_1^1 \cdots v_{m_1}^1, \ldots, v_1^n \cdots v_{m_n}^n)$ *independent* if the number of edges (v_i^j, v_{i+1}^j) with $1 \leq i \leq m_j$ and $1 \leq j \leq n$ in this list does not exceed the number of edges $(v_i^j, v_{i+1}^j)_a$ in E $(a \in \Sigma)$, i.e., for each pair $(u, v) \in V \times V$ it holds

$$\left|\left\{(i,j) : (v_i^j, v_{i+1}^j) = (u,v)\right\}\right| \leq \left|\left\{a \in \Sigma : (u,v)_a \in E\right\}\right|.$$

Based on this definition we now define schemes for dynamic network routing games with simple constraints. A *scheme* for a routing game $\mathcal{G} = (G, D, \#e, \#t)$ with $G = (V, E)$ and $D = ((s_1, d_1), \ldots, (s_n, d_n))$ is a function S that maps each set $B \subseteq E$ of blocked edges (with $|B| \leq \#e \cdot \#t$) to a list of independent paths $(\rho_1, \ldots, \rho_n)$ such that ρ_i leads from s_i to d_i for every $i \in \{1, \ldots, n\}$. We denote with $S(B)_i$ the i-th path ρ_i in $S(B)$. We assume that each path ρ_i of a scheme does not have any loop and hence has a length of at most $|V|$. Then, there exist only finitely many schemes S for a game $\mathcal{G}$. Consequently, we can compute all possible schemes S with an exhaustive search.

Concerning the complexity of computing a scheme, let us note that one has to compute independent paths from the source nodes to the destination nodes for each graph resulting from the connectivity graph by demand agent's edge blockings. Computing these paths for each of these graphs is an *integer multi-commodity flow problem*, which is known to be NP-complete (Even, Itai, and Shamir, 1976; also see Costa, Létocart, and Roupin, 2005). However, one has to find independent paths for each set of blocked edges B with $|B| \leq \#e \cdot \#t$. The number of different sets containing at most $\#e \cdot \#t$ elements is $\sum_{k=0}^{\#e \cdot \#t} \binom{|E|}{k} = 2^{\#e \cdot \#t}$. However, one could formulate the routing algorithm also in a way where routing agent always assumes that exactly $\#e \cdot \#t$ edges are blocked in every turn. More precisely, if demand agent blocks a set B containing less than $\#e \cdot \#t$ edges, there

also exists a set B' with $B \subseteq B'$ containing exactly $\#e \cdot \#t$ edges. Clearly, if there exist independent paths in the graph resulting from removing the edges in B', these independent paths also exist in the graph resulting from removing the edges for B. In this way one obtains a slightly better bound since there are only $\binom{|E|}{\#e \cdot \#t}$ different sets containing exactly $\#e \cdot \#t$ blocked edges.

Using Schemes for Routing in Boundedness Games

We use a scheme by routing packets via their independent paths in $S(B)$ where B is the set of edges that were blocked when the packets are generated. Clearly, as long as the set of blocked edges B does not change, routing agent delivers every packet in at most $|V|$ turns. When the set of blocked edges changes, say from B to B', also the independent paths change from $S(B)$ to $S(B')$. So, in general some undelivered packets in the network cannot be routed anymore. In this case, our routing algorithm only routes packets that were generated while the edges in the current set B' were blocked. However, when the set B is blocked again, routing agent continues to deliver the previously generated packets. Therefore, the number of packets in the network stays bounded.

To realize this procedure, we have to remember which edges were blocked when a packet was generated. For this reason, we denote with $\mathit{PastB}(t)$ the set B of edges that were blocked t turns before (so, $\mathit{PastB}(0)$ denotes the set of the currently blocked edges). It is easy to prove that the existence of a scheme implies a solution to the boundedness game.

Theorem 2.24. *Let $\mathcal{G} = (G, D, \#e, \#t)$ be a dynamic network routing game with simple constraints and $D = ((s_1, d_1), \ldots, (s_n, d_n))$. If there exists a scheme for $\mathcal{G}$, routing agent wins the boundedness game $\mathcal{G}$.*

Proof. Assume that there exists a scheme S for $\mathcal{G}$. We define a strategy for routing agent as follows. In every turn where currently the set of edges B is blocked, routing agent sends every packet at u, say (id, s_j, d_j, t), to the next node on the path $S(B)_j$ if $B = \mathit{PastB}(t)$; otherwise she does not send the packet. In the case that the path $S(B)_j$ is ambiguous due to multiple occurrences of the pair (s_j, d_j) in D, routing agent ensures that every packet is routed via the same path as long as the same set B of blocked edges is blocked. Note that there are at most as many packets of the form

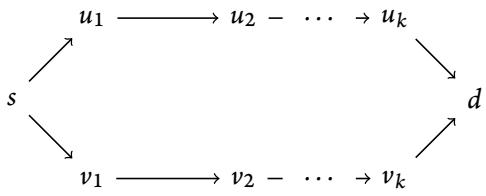

Figure 2.2: A family G_k of connectivity graphs.

(id, s_j, d_j, t) at a node u as there are paths $S(PastB(t))_j$. Since the used paths are independent, the described routing actions are always possible.

By playing the proposed strategy, routing agent guarantees that – for every set of blocked edges – there are at most $|V|$ packets on each path (since the length of each paths is at most $|V|$). As we have n pairs (s_j, d_j) of source and destination, which are connected via independent paths, the number of undelivered packets on these paths is bounded by $n \cdot |V|$. In the worst case this amount of undelivered packets in the network accumulates for each possible set B of blocked edges (with $|B| \leq \#e \cdot \#t$). The number of different sets containing at most $\#e \cdot \#t$ edges can be bounded by $\sum_{k=0}^{\#e \cdot \#t} \binom{|E|}{k} = 2^{\#e \cdot \#t}$.[8] So, overall, there are at most $n \cdot |V| \cdot 2^{\#e \cdot \#t}$ packets in the network in any turn. □

The proof of the previous theorem is constructive in the sense that a positional winning strategy for routing agent can be computed if there exists a scheme. However, her winning strategy may depend on the blocked edges in the entire network. In general this is unavoidable. For instance, consider the family of directed connectivity graphs $\{G_k\}_{k \in \mathbb{N} \setminus \{0\}}$ depicted in Figure 2.2 and the boundedness games $\mathcal{G}_k = (G_k, D, 1, 1)$ with $D = ((s, d))$. Demand agent can block at most one edge in every turn. Hence, routing agent wins the boundedness game $\mathcal{G}_k$ for every k by routing the packets either via the path $su_1 \cdots u_k d$ or via the path $sv_1 \cdots v_k d$ depending on which of these paths all edges are available. It is also easy to see that a winning strategy for routing agent must depend on the $(k+1)$-hop neighborhood. More precisely, to route a packet generated at the node s routing agent has to know whether the edge (u_k, d) or (v_k, d) is blocked.

8 As mentioned before, if routing agent assumes that exactly $\#e \cdot \#t$ edges are blocked in every turn, we can improve this bound to $\binom{|E|}{\#e \cdot \#t}$.

However, this example only works for directed edges. In the undirected case routing agent would be able route the packets first via the path $su_1 \cdots u_k d$ as long as the edges on this path are available. If demand agent blocks one of the edges on this path, say (u_i, u_{i+1}) or (u_i, d), routing agent can extend the path to $su_1 \cdots u_i ui - 1 \cdots u_1 sv_1 \cdots v_k d$, which in this case does not contain any blocked edge. By routing packets in this way we obtain a winning strategy for routing agent. Note that this only works because routing agent can use an edge u, v for two transmissions in the same turn, one from u to v and one from v to u. Here it remains open whether routing agent always has a completely local strategy for boundedness games on undirected connectivity graphs, as we are interested in a general purpose solution.

A Routing Algorithm for Boundedness Games

Now we formulate a routing algorithm based on Theorem 2.24. However, the strategy provided with the proof of this theorem has the disadvantage that routing agent sends a packet with time stamp t only if the set $PastB(t)$ coincides with the set B, i.e., if the edges that were blocked t turns before coincides with the currently blocked edges. This means that the algorithm would need to memorize the sets $PastB(t)$ for all previous turns, which is unsuitable for implementation as plays are infinite. To avoid this problem we propose an algorithm that only checks whether there exists a packet with source s_j and destination d_j on the path $S(B)_j$ and, if this is the case, sends the packet to the next node on the path $S(B)_j$. In contrast to Theorem 2.24 this may cause that some packets that have been partially routed via some path $S(B)_j$ may be routed via some path $S(B')_j$ when the set of blocked edges changes from B to B'. This affects the delivery time of the packets but does not increase the upper bound on the number of packets in the network.

The proposed routing algorithm is implemented in Algorithm 2.3, which runs locally on every network node. As input it only needs the name of its own node and the scheme that has been precomputed with respect to the given dynamic network routing game. The algorithm guarantees that the number of packets in the network stays bounded whenever routing agent wins the boundedness game. The algorithm also guarantees that the generated packets for each pair of source and destination are

Algorithm 2.3: A routing algorithm based on a scheme S that keeps the number of packets bounded under simple constraints.

Input: routing game $\mathcal{G}$ with simple constraints, own node name u, routing scheme S

```
1 while true do (* repeat forever *)
2   get set B of blocked edges
3   for each j in {1, ..., |D|} do
4     if the path S(B)_j = u_1 ··· u_m contains the current node u, and
      there exists a packet (id, s, d, t) at u with s = u_1 and d = u_m then
5       route the packet (id, s, d, t) with the highest time stamp t
        to the next node on the path S(B)_j
6     end if
7   end for
8   wait for the next turn
9 end while
```

routed via the same path as long as the same set of edges is blocked. When the set of blocked edges changes, however, this property may be violated for several turns.

Limitations of Routing via Schemes

As mentioned before, we focus here on routing algorithms that forward packets generated due to the same pair of source and destination via the same path (as long as the same edges are blocked). It is easy to see that a scheme is a requirement for a solution satisfying our additional demand.

Remark 2.25. Let $\mathcal{G} = (G, D, \#e, \#t)$ be a dynamic network routing game with simple constraints and $D = ((s_1, d_1), \ldots, (s_n, d_n))$. Assume that there does not exist any scheme for $\mathcal{G}$. Even under the assumption that demand agent always blocks the same set of blocked edges (from some turn onwards), routing agent does not have a strategy to win the boundedness game $\mathcal{G}$ by routing the packets that were generated for each pair of source and destination via the same path.

Proof. Assume that routing agent has such a strategy, but that there does not exist a scheme for $\mathcal{G}$. This means that demand agent can block a

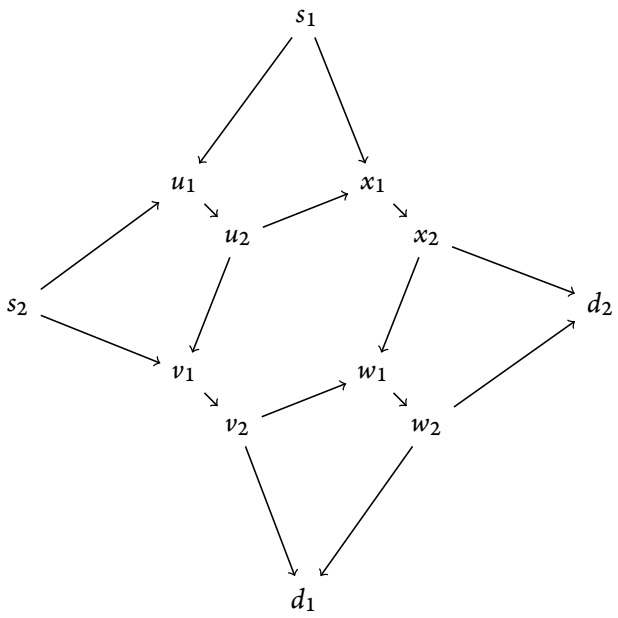

Figure 2.3: A connectivity graph of a boundedness game which routing agent wins, but for which a scheme does not exist.

set B of edges with $|B| \leq \#e \cdot \#t$ such that in the resulting graph, say G', there do not exist independent paths connecting all pairs of source and destination. We assume that demand agent always blocks this set B from some turn onwards. But since we assumed that routing agent can keep the number of packets in the network bounded by forwarding the packets generated for each pair of source and destination via the same path, she has to route the packets via independent paths in G'. This is contradiction, because there do not exist independent paths connecting all sources and destinations. □

So, the existence of a scheme characterizes boundedness games where routing agent can win by forwarding packets generated due to the same pair of source and destination via the same path as long as the same edges are blocked. However, there exist boundedness games which routing agent wins but for which do not exist any scheme. In these games, routing agent has to distribute packets generated for some pair of source and destination via different paths even if demand agent blocks the same set

of edges in every turn. As an example, imagine the boundedness game $\mathcal{G} = (G, D, \#e, \#t)$ on the connectivity graph depicted in Figure 2.3 with $D = ((s_1, d_1), (s_2, d_2))$ and $\#e = \#t = 0$. In this game, demand agent cannot block any edges, and it is easy to see that there does not exist any scheme. Nevertheless, routing agent wins the boundedness game by forwarding every packet generated at s_1 (s_2) in an odd turn via the path $s_1u_1u_2v_1v_2d_1$ ($s_2u_1u_2x_1x_2d_2$) and every packet generated at s_1 (s_2) in an even turn via the path $s_1x_1x_2w_1w_2d_1$ ($s_2v_1v_2w_1w_2d_2$). It is an open problem to find a more general concept than schemes that characterizes exactly the solutions of boundedness games with simple constraints.

2.4.3 Bounded Delivery Games

Finally, we propose a routing algorithm for bounded delivery games, which is based on our routing algorithm for boundedness games. We also infer under which condition the algorithm guarantees ℓ-delivery. In order to reuse our scheme-based approach, we introduce additional assumptions for that we can guarantee that routing agent can also deliver every packet (besides guaranteeing boundedness).

Generally speaking, it is very challenging to develop a feasible routing algorithm for bounded delivery games with simple constraints. We cannot use our technique from Section 2.4.1 for solving a delivery game, where only one packet is routed at once. It inherently conflicts with routing all packets simultaneously via schemes, which we used to solve boundedness games. Indeed, bounded delivery games are solvable under simple constraints since we showed that these games are already solvable under weak constraints; but an exhaustive search of the state space does in general not yield an efficient routing algorithm that runs locally on each node.

Therefore, we extend our algorithm for solving boundedness games based on schemes (which we introduced in the previous section). In principle, this is possible because a solution for a boundedness games can only violate a bounded delivery condition with respect to finitely many packets. However, to establish this solution, we need two additional ingredients. On the one hand we require a fairness assumption on demand agent's actions of edge blocking. On the other hand the schemes must have a special property – namely an order on the edges along all routing paths – to yield a suitable solution.

In the following we first introduce the mentioned fairness assumption. Then, we define ordered schemes. Based on these concepts we finally present a routing algorithm that works locally at each network node.

Fair Demands

The main obstacle to apply a scheme (as introduced in Section 2.4.2) to a bounded delivery game arises from the situation where a packet is routed via a node that is suddenly cut off from its destination. Here, we explore a scenario where this cannot happen. We propose the fairness assumption that – for each pair (u, v) of adjacent nodes in the connectivity graph – some edge from u to v is available again and again. Formally, given a routing game $\mathcal{G}$ on a connectivity graph $G = (V, E)$, a strategy σ of demand agent is *fair* if in each play where demand agent plays according to σ the following holds: for each $(u, v)_a \in E$ there is a non-blocked edge from u to v again and again. We say that *demands are fair* if demand agent only plays according to a fair strategy. For bounded delivery games, we also want to be able to determine a delay bound, i.e., a bound on the number of turns that a packet needs to reach its destination. For this reason we also consider the scenario that some edge between two adjacent nodes u and v is available at least every p turns. So, a strategy σ of demand agent is *p-fair* if it guarantees that for each $(u, v)_a \in E$ a non-blocked edge from u to v is available at least every p turns. We say that *demands are p-fair* if demand agent only plays according to a p-fair strategy.

Before we propose a solution for delivery games where demands are fair we note that in this scenario the bounded delivery winning condition and the ℓ-delivery winning condition are in general not equivalent (unlike it was shown by Theorem 2.3 for scenarios where demand agent's strategies are not required to be fair). Of course, Remark 2.1 is still valid, i.e., if routing agent wins a play π with respect to an ℓ-delivery condition, she also wins π with respect to the bounded delivery condition. But when demands are fair, a winning strategy for routing agent in the bounded delivery game does in general not imply that there exists an ℓ such that routing agent wins the ℓ-delivery game. The reason is that demand agent may cut off some packets from their destinations for an increasing number of turns. Then, the packet delay (delivery time) cannot be bounded, although routing agent wins the bounded delivery game.

Ordered Schemes

Although fair demands guarantee that an edge between each connected pair of nodes is available again and again, there is no guarantee that a certain set B of blocked edges will be blocked again in the future. For this reason it is an insufficient strategy to hold packets that where generated when set B was blocked until B is blocked again and all paths in $S(B)$ become available again. As a solution we will route packet also via their original path in $S(B)$ when the set B is not blocked (but the next edge of the path is available). This can delay routing of other packets in the network. As we will see later in an example, this could lead to a situation where packets on two paths delay each other again and again, so that the number of packets does not stay bounded. This situation, however, cannot occur when the edges on all paths defined by a scheme S obey some (arbitrary) order. In other words, with an ordered scheme S it is not possible to build a loop by concatenating any path fragments defined by S. Formally, a scheme S is called *ordered* if there exists a total order $\preceq$ on the edges (defined by E) such that for every path $S(B)_j = v_1 v_2 \cdots v_k$ with $|B| \leq \#e \cdot \#t$ and $j \in \{1, \ldots, |D|\}$ we have $v_1 \preceq v_2 \preceq \cdots \preceq v_k$. If there does not exist such a total order $\preceq$ on the edges, we call S *non-ordered*.

For an example, on the connectivity graph depicted in Figure 2.4, consider the routing game $\mathcal{G} = (G, D, \#e, \#t)$ with $D = ((s, d))$, $\#t = 1$, and $\#e = 2$. We consider the sets $B_1 = \{(s, u_1), (v_2, d)\}$ and $B_2 = \{(s, v_1), (u_2, d)\}$ of blocked edges. To treat these scenarios, we could define a scheme S which defines the paths

$$S(B_1) = (s v_1 v_2 u_1 u_2 d) \quad \text{and} \quad S(B_2) = (s u_1 u_2 v_1 v_2 d) \ .$$

Clearly, S is *non-ordered*, because for the total order $\preceq$ we would obtain $v_1 \preceq u_1$ and $u_1 \preceq v_1$, which would be a contradiction as $u_1 \neq v_1$. Of course, in this example, one could also define an ordered scheme, for instance, one where packets are always tried to be routed directly from s to d. For a connectivity graph where this edge does not exist, this is not possible; in the resulting game, however, independent paths do not exist at all against all possible sets of blocked edges. It is an open problem, whether there exists a routing game with simple constraints for which there exists only a non-ordered scheme.

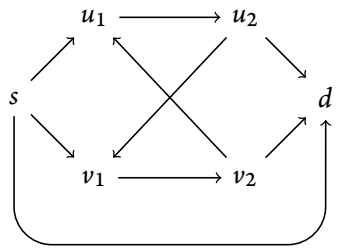

Figure 2.4: A connectivity graph of a bounded delivery game illustrating the difference between ordered and non-ordered schemes.

Using Schemes for Routing in Bounded Delivery Games

In the following we propose a simple routing strategy based on ordered schemes; it is a refinement of the routing algorithm that we introduced in Section 2.4.2 for boundedness games with simple constraints. The basic idea is to send each packet via the independent paths $S(B)$ as long as the same set of edges B is blocked. When the set of blocked edges changes, say from B to B', routing agent routes all the newly generated packets via the paths $S(B')$ and tries to route the previously generated packets via the previous paths $S(B)$. This is eventually possible since an edge between two adjacent nodes will be available again and again. Nevertheless, such a procedure involves the problem that the packets which remain to be routed via previous paths may cause a delay in the packet delivery. We show that the number of packets in the network still stays bounded if the used scheme is ordered.

In detail, we propose the following routing strategy in a bounded delivery game $\mathcal{G} = (G, D, \#e, \#t)$ for which we assume that there exists an ordered scheme S and that demands are fair. In every turn routing agent tries to send every packet at u, say (id, s_j, d_j, t), beginning with the highest time stamp t to the next node on the path $S(PastB(t))_j$. In the case that the path to choose from $S(PastB(t))$ is ambiguous due to multiple occurrences of the pair (s_j, d_j) in D, routing agent routes every packet (id, s_j, d_j, t) always via the same path $S(PastB(t))_j$, i.e., she assigns a fixed index j to each packet when it is created.

Theorem 2.26. *Let $\mathcal{G}$ be a dynamic network routing game with simple constraints, and assume that exists an ordered scheme S for $\mathcal{G}$. If demands are fair,*

routing agent wins the bounded delivery game $\mathcal{G}$. Moreover, if demands are p-fair, demand agent wins the ℓ-delivery game for $\ell = p \cdot |D| \cdot (2^n - 1)$, where n denotes the number of single edges in the network graph.[9]

Proof. We show that the claims hold with the aforementioned routing strategy. Clearly, with the proposed strategy routing agent eventually delivers every packet (because packets with a higher time stamp are routed with a higher priority and a link between each pair of adjacent nodes is available again and again).

In the following we show that also the number of packets in the network stays bounded. Since S is an ordered scheme, there exists an order on the edges, say $e_1 \preceq \cdots \preceq e_n$, that is respected by every path defined by S. Let k_i^t be the number of packets in the network at turn t that still have to be routed via the edge e_i. We claim that $k_i^t \leq k_i$ where $k_1 := |D|$ and $k_{i+1} := |D| + \sum_{j=1}^{i} k_j$. This claim implies that – in every turn – the number of packets in the network is bounded, for instance, by $\sum_{i=1}^{n} \cdot k_i \leq \sum_{i=1}^{n} 2^{i-1} \cdot |D| \leq n \cdot 2^{n-1} \cdot |D|$. We prove the claim by induction over the turn number t. In the first turn $t = 1$, the claim is obviously true (as demand agent can generate at most $|D|$ packets in one turn).

For the induction step, let us assume that the claim holds in turn t, and let us consider some edge $e_i = (u, v)$ with $i \in \{1, \ldots, n\}$. Let m_i be the number of packets that are generated in turn $t + 1$ and that have to be routed via an edge e_i. We distinguish between two cases depending on the previous value k_i^t. If $k_i^t \leq k_i - m_i$, it clearly holds $k_i^{t+1} \leq k_i^t + m_i \leq k_i$. Otherwise we have $k_i^t = k_i - m_i + x$ for some $x \in \{1, \ldots, m_i\}$. For the latter case, we now show that there are at least x packets at u. All packets that still have to be routed via e_i and that are not at u have to be routed at least via some other edge e_j with $j < i$ (as $e_j \preceq e_i$). We know by induction that $k_j^t \leq k_j$ for all $j \in \{1, \ldots, n\}$. We conclude that the number of packets in turn t that have to be routed via e_i but that are not at u is at most $\sum_{j<i} k_j^t \leq \sum_{j<i} k_j \leq k_i - |D| \leq k_i - m_i$. This shows that there are at least x packets at u in turn t. Because m_i packets are generated in turn $t + 1$ that have to be routed via the edge e_i, the multiplicity of the edge must be at least m_i. Since $x \leq m_i$, routing agent can forward the x packets at u via e_i. Hence, in turn $t + 1$, the number of packets to be routed via e_i remains bounded by k_i.

9 The proof idea for this theorem is due to Christof Löding.

It remains to be shown that routing agent can deliver each packet within $p \cdot |D| \cdot (2^n - 1)$ turns if demands are p-fair. To show this, we note that in each turn the number of packets that still have to be routed via the edge $e_i = (u, v)$ is bounded by $k_i \leq |D| \cdot 2^{i-1}$. Therefore, an individual packet that is waiting at u can be delayed by at most $|D| \cdot 2^{i-1}$ packets. As a consequence, routing agent sends this packet to v in at most $p \cdot |D| \cdot 2^{i-1}$ turns, because at least every p turns an edge connecting u to v becomes available. Since – in the worst case – routing agent transmits the packet via all of the edges $e_1, \ldots, e_n$, each packet reaches its destination within $\sum_{i=1}^{n} p \cdot |D| \cdot 2^{i-1} = p \cdot |D| \cdot (2^n - 1)$ turns. □

A Routing Algorithm for Bounded Delivery Games

For the sake of completeness, we now formulate the proposed routing algorithm in more detail. This algorithm requires that there exists an ordered scheme S for the given scenario. Then, it works for bounded delivery games under the assumption that demands are p-fair. The routing algorithm tries to route each packet (id, s_j, d_j, t) via its path $S(PastB(t))_j$. For this reason the routing algorithm has to save the sets of the previously blocked edges $PastB(t)$ for every $t \in \{1, \ldots, \ell\}$, where $\ell = p \cdot |D| \cdot (2^n - 1)$ is the worst-case delay bound of the packets as provided by Theorem 2.26. If a pair (s_j, d_j) occurs multiple times in D, we have to guarantee that each packet (id, s_j, d_j, t) is always routed via the same path $S(PastB(t))_j$. For this reason we assume that the set of possible identifiers (i.e., the set of natural numbers) is partitioned into $|D|$ sets, each of which containing the identifiers for the packets that demand agent generates for the j-th pair in D. For instance we can assume that, for each identifier id_j of some packet generated for the j-th pair in D, it holds

$$id_j \bmod |D| = j\,.$$

This concept is implemented in Algorithm 2.4. The algorithm fetches the set B of the currently blocked edges in every turn and updates the sets $PastB(t)$ accordingly. Then it tries to route each packet (id_j, s, d, t) via the j-th path in $S(PastB(t))$, where id_j is an identifier associated with the j-th pair in D. Under the assumption that demands are p-fair, the algorithm, which runs locally on each network node, guarantees that every packet is delivered within $\ell = p \cdot |D| \cdot (2^n - 1)$ turns.

Algorithm 2.4: A routing algorithm based on an ordered scheme S that delivers each packet within a bounded delay for p-fair games under simple constraints.

Input: routing game $\mathcal{G}$ with simple constraints, own node name u, ordered routing scheme S, worst-case delay bound ℓ

```
1  while true do (* repeat forever *)
2     get set B of blocked edges
      (* update the sets of the previously blocked edges *)
3     for t from 1 to ℓ do
4        PastB(t) := PastB(t − 1)
5     end for
6     PastB(0) := B
      (* route packets *)
7     for each packet (id_j, s, d, t) (beginning with the highest time stamp)
      do
8        set ρ to the j-th path in S(PastB(t))
9        send the packet (id_j, s, d, t) to the next node on the path ρ if
         an accordant edge is still available for transmission
10    end for
11    wait for the next turn
12 end while
```

If we only assume that demands are fair (but not p-fair), the algorithm cannot be used in the stated form. The reason is that we do not know a worst-case delay bound ℓ, which we used to bound the number of turns for that we must remember the previously blocked edges. Moreover, it is possible that such a worst-case delay bound does not even exist. One can imagine different solutions to overcome this issue. For instance, one can think of a variant of the routing algorithm that can use infinite memory or – as an approximation to this – saves the previously blocked edges for a very long time. In practice, this could be an adequate approach, since usually one does not care anymore about packets that were not delivered in a reasonable time. Another approach would be to broadcast the ages of the packets through the network. Then, the routing algorithm can discard all older sets of previously blocked edges. At least the number of needed

communication messages is bounded, because we have shown a bound on the number of packets in the network in Theorem 2.26. However, a mechanism to guarantee that each broadcast eventually reaches every node is required.

2.5 Conclusion and Open Problems

In this chapter we introduced dynamic network routing games and studied various routing problems, each of which corresponds a particular winning condition: delivery, boundedness, bounded delivery, or ℓ-delivery. As principle limitations we showed that – in the general framework – ℓ-delivery games are algorithmically solvable, whereas solving boundedness, delivery, and bounded delivery games is undecidable. With weak constraints we presented a coarser model; it prevents that the network packets can be used as a memory structure to do computations. We showed that, under weak constraints, solving routing games is decidable for all of our winning conditions. To obtain – beyond the principle solvability – concrete routing algorithms, we studied routing games with simple constraints, which are again coarser than weak constraints. For delivery games with simple constraints, we showed that we can always put a winning strategy into a simple routing algorithm, which runs locally on each network node. For delivery games where demand agent claims the blocking of edges for the next turn only, we proposed a variant of this algorithm where even the precomputations for the routing decisions can be done in linear time. For boundedness games, we needed an additional requirement to transform a winning strategy into a simple routing algorithm; namely, we assumed that the forwarding path of each pair of source and destination stay fixed as long as the blocked edges stay fixed. Finally, we extended this routing algorithm to solve bounded delivery games; for this, we required (besides the aforementioned assumption for boundedness games) an additional fairness assumption on the behavior of demand agent. In each of the proposed algorithms for games with simple constraints, a routing decision at a node only requires knowledge of a local neighborhood of this node.

Although we provide routing algorithms which run efficiently once some precomputations for the given routing game have been done, this chapter only sketches the complexity issues of solving routing games and

computing winning strategies. For solving routing games under weak constraints, and also for solving ℓ-delivery games in the general setting, our decidability results only provide very expensive methods. The bounds on the state space we provide lead to a multi-exponential number of positions in the unfolding (on which we then solve a safety game that is equivalent to the original routing game). It is an open problem to find tighter bounds for the complexity of solving these routing games. Also bounds on the size of the obtained winning strategies are still missing. It is also unknown whether computing a winning strategy is as hard as determining the winner in a routing game.

Problem 2.1. For the decidable cases, prove tight upper and lower complexity bounds for determining the winner of a dynamic network routing game. Which computational effort is needed to compute a winning strategy for the winning agent? How much memory is needed to represent a winning strategy?

For solving delivery games under simple constraints we already obtained tighter complexity bounds. We can solve these games efficiently in the case of $\#t = 1$, i.e., in the case that demand agent completely determines the set of blocked edges in every turn. Then, we can compute a winning strategy in linear time. For solving delivery games under simple constraints in general, we have to solve extended sabotage games for all pairs of source and destination. Solving extended sabotage games is at least PSPACE-hard (since solving the original sabotage games is PSPACE-hard as discussed in Chapter 1); but so far we only know an EXPTIME upper bound. To improve this bound to PSPACE it would suffice improve the bound in Lemma 2.21 (or in Corollary 2.22); more precisely, one needs to prove a polynomial bound on the number of times that Runner needs to revisit a vertex (or on the number of turns that Runner needs to win) in an extended sabotage game.

Problem 2.2. Does there exist a polynomial bound on the number of turns in which Runner visits a certain vertex in an extended sabotage game (and hence a polynomial bound on the number of turns that Runner needs to win an extended sabotage game)? Are extended sabotage games solvable in polynomial space?

The routing algorithms that we proposed for boundedness games (and bounded delivery games) with simple constraints require (ordered) schemes; these schemes maps each possible set of blocked edges to independent paths. Finding independent paths is an *integer multi-commodity flow problem*; this can be formulated as an *integer programming problem* and is known to be NP-complete (Even, Itai, and Shamir, 1976; also see Costa, Létocart, and Roupin, 2005). However, in practice one should obtain a solution to this problem in reasonable time, especially if the length of the paths (the number of hops) and the maximal number of adjacent nodes (the node degree) are small. It is also possible to restrict the allowed number of hops in order to obtain shorter routing paths and decrease the computation time. The more crucial aspect of computing schemes is that one has to find independent paths for each set B of blocked edges with $|B| \leq \#e \cdot \#t$. We already mentioned that one can reduce this effort by only considering *maximal edge blockings*, i.e., edge blockings with $|B| = \#e \cdot \#t$. In practical scenarios, however, the number of combinations of blocked edges may be much lower. For instance, one can consider a fixed number of interferers, each of which only blocks a certain subset of the edges. Then, one needs to compute independent paths only for the edge blockings that can actually be caused by these interferers.

Another aspect of computing schemes is that independent paths needs to be computed for very similar sets of blocked edges. Even if one edge is available instead of another one, all independent paths have to be recomputed with the naive approach. It is unknown whether one can save some of the computational effort. This also raises the question whether a scheme for a routing game $\mathcal{G}$ can be adapted if we change the routing game by adding an edge to the connectivity graph or a pair of source and destination. It is an ambitious task to develop an online algorithm that efficiently maintains a scheme (and maybe some auxiliary data structure) while small updates are committed to the considered routing game.

Problem 2.3. Does there exist a more efficient way to compute independent paths for a maximal edge blocking if independent paths have been already computed for other edge blockings? Does there exist an online algorithm for computing schemes, so that the computational effort for each maintenance step is significantly lower than the effort for recomputing the scheme?

Our scheme-based routing algorithm for boundedness games only depends, at each node u, on the packets at u and on the blocked edges in the network. Although we showed that the knowledge about the blocked edges in the network cannot be restricted in general, there exist many routing games where this is possible. For boundedness games on undirected connectivity graphs, we discussed that it seems possible to route packets at each node only depending on which incident edges are blocked (as in the example sketched in Figure 2.2). It remains to be proven whether such a completely local routing scheme exists for every boundedness game (with simple constraints) on undirected connectivity graphs and how it can be computed.[10]

One might also consider to restrict schemes in general in a way that the paths can be uniquely determined at each node u depending on the blocked edges in the k-hop neighborhood of u. Whenever such a *k-local scheme* exists, the routing agent should have a winning strategy that only depends on the k-hop neighborhood of each node. However, k-local schemes seems to be hard to compute, as one has to ensure that each independent path can be uniquely determined with local knowledge.

Problem 2.4. Consider a boundedness game with simple constraints on an undirected connectivity graph. If routing agent wins, does she always win with a routing strategy that only depends, at each node u, on the packets at u and the incident edges of u? In general, try to develop a notion of *k-local schemes* where paths can be uniquely determined at each node by knowledge of the edges in the k-hop neighborhood only. Does there exist a local routing algorithm based on these k-local schemes whose routing decisions at each node can be computed efficiently (and only depend on the k-hop neighborhood)? Which computational effort is needed to compute a k-local scheme?

We have seen that the solutions via schemes do not capture all solutions of boundedness games under simple constraints. In some boundedness games (as the one sketched in Figure 2.3), it is necessary to route

10 However, in a more involved network model where routing agent can transmit from each node $v \in V$ only one packet per frequency $a \in \Sigma$ (as defined in Gross, Radmacher, and Thomas, 2010), this observation does indeed not hold anymore. In this case sending a packet back via a path (via which it was send before) does not come for free, because each transmission consumes a frequency for all packets at some node.

packets of the same type via different paths even if the set of blocked edges does not change. For this reason, solving a boundedness game seems to be closer related to a *fractional multi-commodity flow problem* (see Shahrokhi and Matula, 1990; Garg and Könemann, 2007). It is an open question whether there is a more appropriate notion of schemes based on these problems. There is hope that boundedness games are then solvable more efficiently, because *fractional* multi-commodity flow problems are solvable by linear programming (see Shahrokhi and Matula, 1990; Garg and Könemann, 2007).

Problem 2.5. Is there another notion of local routing schemes that capture all solutions of boundedness games? (Of course, this new notion of routing schemes does not need to follow the restriction that forwarding paths stay fixed as long as the blocked edges do not change.) Which computational effort is needed to compute such a scheme? Is it possible to compute the routing decisions for these new kind of routing schemes by solving *fractional* multi-commodity flow problems?

We used ordered schemes for solving bounded delivery games under simple constraints. For this, our routing algorithm required that demands are fair, besides the assumptions that routing agent has a winning strategy and that she has to forward packets via the same path as long the same edges are blocked. We illustrated (by the routing game sketched in Figure 2.4) that our routing algorithm needs ordered schemes to work probably. However, it is an open problem whether the requirement for open schemes lessens the number of games that we can solve with our routing algorithm. An example of a bounded delivery game where only exists a non-ordered scheme (but not an ordered scheme) is still missing.

Problem 2.6. Let us restrict ourselves to bounded delivery game with simple constraints where demands are fair and where routing agent has a winning strategy which forwards all packets that were generated for each pair of source and destination via the same path as long as the set of blocked edges does not change. Does there exists a game satisfying all of the mentioned requirements for which there exists an ordered scheme but not a non-ordered scheme?

Also for bounded delivery and ℓ-delivery games a notion of routing schemes and local routing algorithms that capture exactly the solutions

of these games are still missing. A first step would be to provide such a local routing algorithm in the case of $\#t = 1$, i.e., in the case that demand agent completely determines the blocked edges in every turn. In this case it might be possible to combine the method of routing packets via some kind of routing schemes with the routing algorithm that we developed for delivery games.

Problem 2.7. Is there another notion of local routing schemes or a local routing algorithm that captures all solutions of bounded delivery or ℓ-delivery games? Is the same possible for the case of $\#t = 1$ (possibly in a more efficient way)? What is the computational complexity of solving bounded delivery and ℓ-delivery games under simple constraints?

At the end of this chapter we illustrate some ways to refine the model and the problem statement for practical applications. One issue that one might observe is that, in delivery, bounded delivery, and ℓ-delivery games, a single undelivered (or lately delivered) packet suffices for demand agent to win the game. In the boundedness games, on the other hand, a fixed number of packets may not be delivered, but we can obtain neither an upper bound on the number of undelivered packets nor a worst-case delay bound (i.e., a bound on the delivery time) for the delivered packets. Therefore, it is natural to allow routing agent to drop packets. Then, routing agent has to ensure the winning condition with respect to the non-dropped packets while guaranteeing that the number of dropped packets does not exceed a given *drop rate*.

Also our routing game model could be criticized to be rigid in the sense that the demand agent blocks edges as fast as the routing agent can transmit packets between adjacent nodes. To handle this problem one can introduce an additional parameter, say a *waiting time* $\#w$, that specifies that demand agent may block edges at most every $\#w$ turns.

Finally, one might also combine our routing game model with stochastic aspects as demands are usually best described in a stochastic model. Especially for the routing game model based on simple constraints, it is easy to model the network traffic stochastically. There, we only have to replace the fixed list of pairs of source and destination by a probability distribution on each pair of network nodes. Also, one might replace the two parameters $\#e$ and $\#t$ by appropriate probability distributions. This turns our two-player routing game into a Markov decision process. Then,

one can determine, for example, the probability with which a worst-case delay bound can be guaranteed for each packet.

Constructor vs. Destructor

3

Dynamic Network Connectivity Games

Complementary to the previous chapters, where we studied routing problems, the subject of this chapter is a game model that focuses on the network connectivity. A *dynamic network connectivity game* is played by two players, called *Destructor* and *Constructor*. While Destructor can delete nodes, Constructor can relabel nodes, restore nodes, or create complete new nodes in the network with respect to a given set of rules.

We distinguish three different types of rules for Constructor, each of which corresponds to one of Constructor's possible action to maintain the network. The first type of rule is concerned only with the information flow through the network (evoked, for instance, by the users of the network); 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. Therefore,

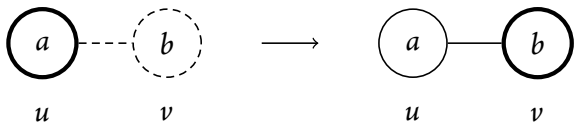

Figure 3.1: Movement of a strong node u restoring a deleted node v.

the rules of the first type are *relabeling rules*, which describe 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. For these operations Constructor 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 also 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 3.1). For the creation of a node we can pick some set U of strong nodes, create a new node v (which may be strong or not) and connect it by an edge with each node of U (see Figure 3.2). Both the movement of a strong node and the creation of a new node 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 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 3.1.

In this chapter we are only concerned 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

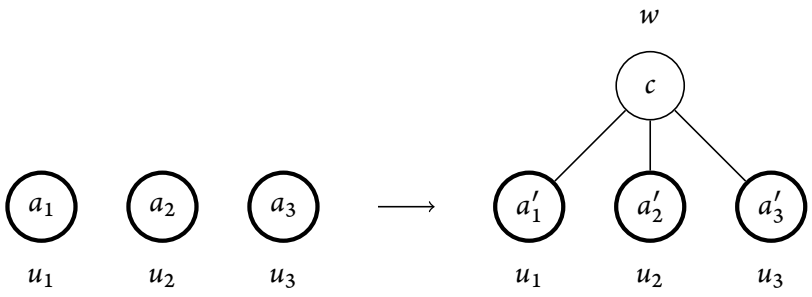

Figure 3.2: Creation of a new node by a set $U = \{u_1, u_2, u_3\}$ of strong nodes.

version, where Constructor has to guarantee that the network is always connected. We analyze the decidability and the computational complexity of solving these safety connectivity games (Section 3.2) and reachability connectivity games (Section 3.3). 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 are restricted or node labels are omitted these games become solvable. Some of our complexity results for the decidable fragments depend on the balance between node deletion and restoration; if Constructor restores a node or creates a new node, Destructor can delete another one immediately. Therefore, we consider also connectivity games with *multi-rules* where multiple movement and relabeling actions can be combined into a single rule. We will show that multi-rules rise the complexity of solving connectivity games for some decidable fragments. At the end of this chapter we summarize our results (see Table 3.1, page 184), and we also discuss some cases in which solving connectivity games remains an open problem (Section 3.4).

Many results in this section were developed in discussion and collaboration with Sten Grüner and Wolfgang Thomas (see Radmacher and Thomas, 2008; Grüner, Radmacher, and Thomas, 2011, 2012). The results on multi-rule games were not published before.

3.1 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

In contrast to the previous chapters where networks were represented as graphs (possibly with multiple edges), connectivity games require an enriched graph structure that captures the features of node labels, deactivated (deleted) nodes, and maintenance resources (strong nodes), which are needed to restore deactivated nodes and to create new nodes. Formally, a *network* is a tuple

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

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*, and
- a partition of V into sets $P_{a_1}, \ldots, P_{a_k}$ 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 for Constructor 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

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

Constructor applies the relabelings one after the other, but in the same move. Moreover, we require that Constructor applies all or none of the relabelings of such rule, i.e., Constructor is only allowed to choose a rule with multiple relabelings if she then carries out each of the relabeling actions.

Movement rule: Each rule $\left\langle a \xrightarrow{\text{move}} b \right\rangle$ 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 3.1. We use both terms "moving a strong node" and "shifting its strongness" in this chapter 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

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

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 3.2.

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

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

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, before this rule can be applied again on U, additional relabeling steps may be required. This corresponds to our intuition that node creation causes higher costs than restoration.

We also consider some variants of the connectivity game in which 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 $\left\langle \sqcup \xrightarrow{\text{move}} \sqcup \right\rangle$ is the only available rule.

We will also consider connectivity games with *multi-rules*; there R is allowed to contain rules that combine multiple of the above introduced rules into a single one (not only in the case of relabelings as in usual connectivity games). For instance the rule

$$\left\langle a \xrightarrow{\text{move}} b \;;\; a, b \longrightarrow c, d \;;\; d \xrightarrow{\text{move}} e \right\rangle$$

allows Constructor to do in one turn first a movement step, then a relabeling step, and finally again a movement step. The parts of such a rule can only be applied together in the same turn and only in the given order. We shall pay attention to non-expanding games with multi-rules, where each rule may consist of arbitrary many steps, but must not involve any node creation; we will show that the computational complexity to solve these games increases if we allow multi-rules.

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 and Determinacy

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. A *strategy for Constructor* is again such a function (denoted by τ) where i is even and G_{i+1} 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. *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.

It is easy to show that every reachability and safety connectivity game is determined, i.e., either Destructor or Constructor has a winning strategy. In the same way as in the games treated in the previous chapters, this follows from the fact that every game with a Borel type winning condition is determined (Martin, 1975; also see Martin, 1985; Kechris, 1995). To apply this result it suffices to transform the connectivity game into an *infinite two-player game played on a graph* (see Thomas, 2008a; Grädel, Thomas, and Wilke, 2002), in which each vertex corresponds exactly to one network of the connectivity game and an indication of which player acts next. Each vertex in the unfolded game graph corresponds either to a connected or a disconnected network. Hence, every Borel type winning condition over the property of being connected carries over to the unfolded game.

In this thesis we study connectivity games only with reachability and safety objectives. For these winning conditions one can show with the same argument of unfolding the connectivity game that both players' winning strategies can be restricted to positional strategies, i.e., if Constructor (Destructor) wins a game $\mathcal{G}$, she (he) also has a positional winning strategy for $\mathcal{G}$. This follows since in the unfolded reachability or safety game the winning player can always win with a positional strategy (see Thomas,

1995, 2008a; Grädel, Thomas, and Wilke, 2002). Therefore, we will always assume in this chapter that strategies are positional.

Proposition 3.1. *Dynamic network connectivity games with a Borel type winning condition over the property to be connected (e.g., reachability or safety connectivity games) are determined. Moreover, in reachability and safety connectivity games the winning player always has a positional winning strategy.*

The Problem of Solving Connectivity Games

This chapter 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 (see Papadimitriou, 1994), 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 3.2 and 3.3 can be adapted to produce a winning strategy.

Example 3.1. 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 3.3. 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

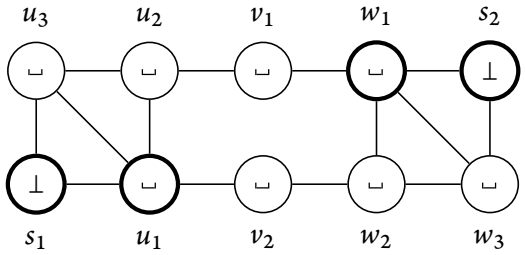

Figure 3.3: The initial network of a safety connectivity game.

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.

First, we consider the rule set R that consists of the rule $\left\langle \text{␣} \xrightarrow{\text{move}} \text{␣} \right\rangle$ 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: $\left\langle \text{␣}, \text{␣} \xrightarrow{\text{create}(\text{␣})} \text{␣}, \text{␣}, \text{␣} \right\rangle$. 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 3.2. 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 3.4. The nodes u_1, v_3, v_6 are deactivated, and the node v_1 is strong. Since this is an unlabeled non-

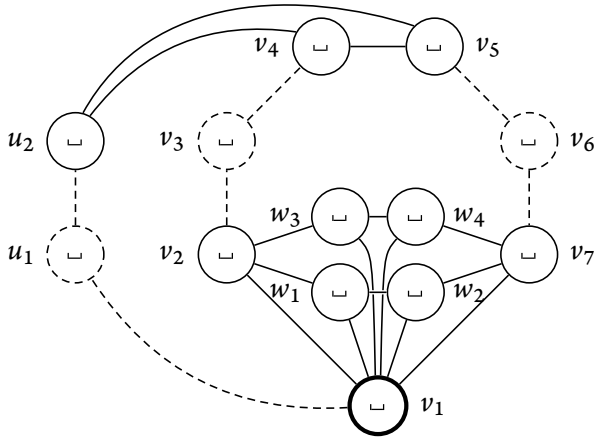

Figure 3.4: The initial network of an unlabeled reachability game.

expanding game, all nodes of G carry the same label $\sqcup$, and the rule set R consists of the single rule $\left\langle \sqcup \xrightarrow{\text{move}} \sqcup \right\rangle$.

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.

3.2 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 3.2.1). Later we point out decidable subcases. We show upper bounds on the complexity of solving these safety games (Section 3.2.2) and a tight lower bound that even holds in the unlabeled non-expanding case (Section 3.2.3). At the end of this section we analyze non-expanding safety games with multi-rules (Section 3.2.4).

3.2.1 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 3.2. *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

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

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 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 $\Gamma \times (\widehat{Q} \cup \{\mathfrak{l}, \mathfrak{r}\}) \times \{\,|\,, \,]\,\}$ with $\widehat{Q} := Q \times \{0, 1, \triangleleft, \triangleright\}$. The first component of each node label holds the content of its represented cell of the tape. The second component is labeled with $\mathfrak{l}$ if the represented cell is on the left-hand side of the head and with $\mathfrak{r}$ if the represented cell is on the right-hand side of the head (the information given by the labeling of the nodes with $\mathfrak{l}$ or $\mathfrak{r}$ 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.

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.

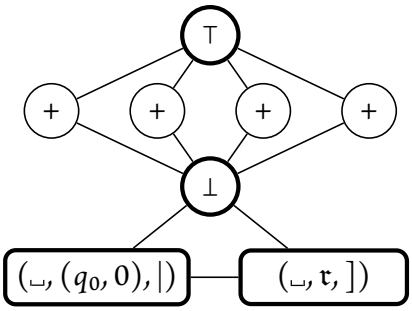

Figure 3.5: The initial network of a safety game representing the initial configuration of a Turing machine.

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 3.5. After some turns of Destructor and Constructor, the network may contain several new cell nodes as well as several new +-labeled nodes. An example of a network after several turns of both players is given in Figure 3.6; 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.

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

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

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 $* \in \{\,|\,, \,]\,\}$:

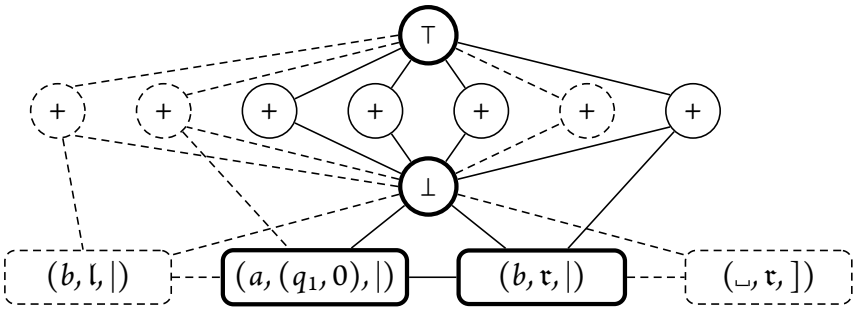

Figure 3.6: A network of a safety game representing a configuration of a Turing machine after several turns of Destructor and Constructor.

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

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

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

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

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

if $X = L$, and

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

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

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

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. □

3.2.2 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 3.8). 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; similarly, we solved sabotage games in Section 1.3.

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 3.3. 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 3.4. *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 has a strict winning strategy σ. 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 3.5. *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 over-approximate 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 3.6. *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 3.7. *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 Thomas, 2008a; Grädel, Thomas, and Wilke, 2002) 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 3.3 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 Thomas, 1995, 2008a; Grädel, Thomas, and Wilke, 2002). □

3.2.3 Unlabeled Non-Expanding Games

We have already showed in Theorem 3.6 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. We show this result by simulating reachability sabotage games, for which we know that solving them is Pspace-hard (Section 1.4).

Theorem 3.8. *In the unlabeled non-expanding case, solving safety connectivity games is Pspace-hard.*

Proof. We use a polynomial-time reduction from solving reachability sabotage games, which is Pspace-hard (see Theorem 1.4), 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 $\left\langle \sqcup \xrightarrow{\text{move}} \sqcup \right\rangle$). For every sabotage game $\mathcal{G}_s = (G_s, v_{\text{in}})$ on a game graph $G_s = (V_s, E_s)$ with a designated set $F \subseteq V_s$ of final vertices, we construct an unlabeled non-expanding game $\mathcal{G}$ such that Constructor can guarantee that the network $\mathcal{G}$ always stays connected iff Runner wins the reachability 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

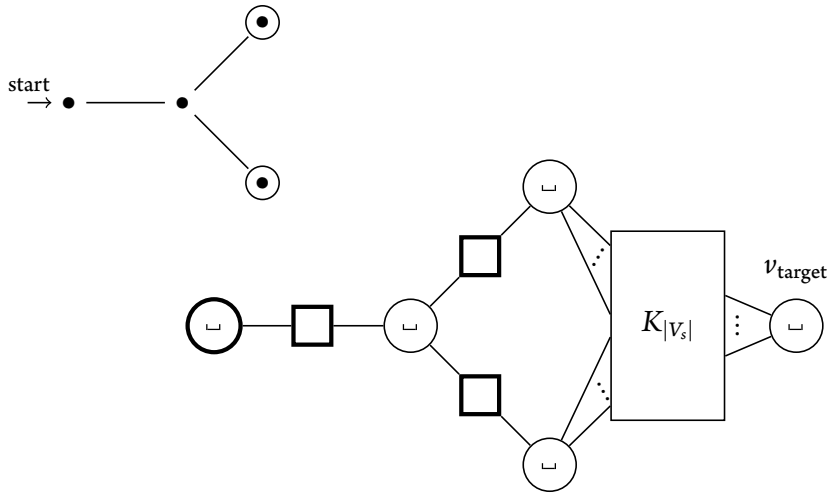

Figure 3.7: A game graph G_s of a sabotage game and its corresponding initial network G of a safety game. Bold squares are the replacement gates for the edges of G_s.

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.

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 3.7 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 3.8. 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 $V := V_s \cup E_s \cup \{w, x_1, x_2, y_1, y_2\} \times E_s \cup \{z_1, z_2\} \cup V(K_{|V_s|}) \cup \{v_{\text{target}}\}$, $S := \{v_{\text{in}}, z_1\} \cup \{w\} \times E_s$, and $P_\sqcup := V$, where $V(K_{|V_s|})$ is the set of

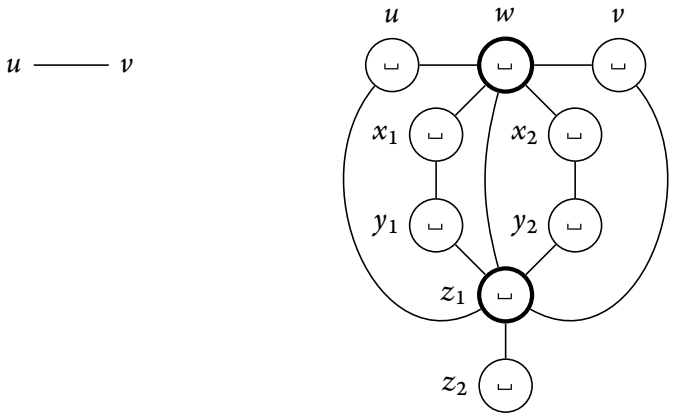

Figure 3.8: An edge between two nodes u and v in the sabotage game and its replacement gate.

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 replacement 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 (Rohde, 2005). Therefore, we assume that Runner wins within $|V_s| - 1$ moves. Whenever Runner moves in $\mathcal{G}_s$ from 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.

3.2.4 Non-Expanding Multi-Rule Games

We have seen that solving non-expanding safety games is PSPACE-complete. This result depends on the balance between node deletion and restoration; if Constructor restores a node, Destructor can delete another one immediately. If we allow rules that combine multiple movement and relabeling actions in a single turn, solving these safety games becomes EXPTIME-complete.

It is easy to see that non-expanding multi-rule games are still solvable in EXPTIME since the unfolding of a non-expanding multi-rule game is still at most exponential in the size of the given game.

Remark 3.9. In the case of non-expanding multi-rule games, solving safety connectivity games is in EXPTIME.

Proof. Analogously to Theorem 3.7 we can transform the game $\mathcal{G}$ into an *infinite two-player game* 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. □

To obtain the ExpTime-hardness, we use a reduction from the halting problem of polynomial space-bounded *alternating Turing machines* (see Chandra, Kozen, and Stockmeyer, 1981; Papadimitriou, 1994). An alternating Turing machine allows besides the existential states of an ordinary non-deterministic Turing machine, also universal states. If the Turing machine is in an existential state, it continues its computation by choosing non-deterministically one of the possible transitions (with the aim to reach eventually an accepting state). If the Turing machine is in a universal state, it has to continue its computation with all of the possible transitions; and for all of these choices the Turing machine must reach eventually an accepting state. We denote with APspace the class of decision problems that can be solved by a polynomial space-bounded alternating Turing machine. We obtain ExpTime-hardness since APspace = ExpTime (see Chandra, Kozen, and Stockmeyer, 1981; Papadimitriou, 1994).

Theorem 3.10. *In the case of non-expanding multi-rule games, solving safety connectivity games is ExpTime-hard.*

Proof. We reduce the accepting problem of polynomial space-bounded alternating Turing machines to the problem of solving non-expanding multi-rule safety games. We present alternating Turing machines in the format

$$M = (Q, \Gamma, \Delta, q_0, g)$$

with a state set Q, tape alphabet Γ (containing a blank symbol $\sqcup$), transition relation $\Delta \subseteq Q \times \Gamma \times Q \times \Gamma \times \{L, R\}$, initial state q_0, and a function $g\colon Q \to \{\wedge, \vee, \text{accept}, \text{reject}\}$ specifying the type of each state. If M is in a state q with $g(q) = \text{accept}$ (with $g(q) = \text{reject}$), M is in an accepting (rejecting) configuration. Configurations where M is in a state q with $g(q) = \vee$ and $g(q) = \wedge$ are existential and universal, respectively. We assume w.l.o.g. that q_{accept} is the only state with $c(q_{\text{accept}}) = \text{accept}$ and q_{reject} is the only state with $c(q_{\text{reject}}) = \text{reject}$. We also assume w.l.o.g. that each configuration, except the accepting and rejecting configuration, has at least

one successor configuration, i.e., for all $(q, a) \in Q \times \Gamma$ with $c(q) = \{\wedge, \vee\}$ there exists a tuple $(q, a, p, b, X) \in \Delta$.

For a polynomial space-bounded alternating Turing machine M and an input word $w \in \Gamma^*$ we construct a game $\mathcal{G} = (G, R)$ such that M does not accept w iff Constructor can preserve the connectivity of the network. The idea is to represent the current configuration of the space-bounded Turing machine by a sequence of nodes. Destructor's objective is to reach a network that represents an accepting configuration; he chooses a transition by actions of node deletion if the Turing machine is in an existential state. Conversely, Constructor's objective is that a configuration containing q_{accept} will be never reached; she chooses a transition by selecting a certain rule from R if the Turing machine is in a universal state. More precisely, we represent a Turing machine configuration $a_1 \dots a_{n-1} q a_n a_{n+1} \dots a_{P(|w|)}$ (where the head is on the a_n-labeled cell) by a sequence of $P(|w|)$ strong nodes labeled with these symbols in succession; per definition of a space-bounded Turing machine the length of the tape of M is bounded by $P(|w|)$ for some polynomial P. Formally, we label these nodes of the network with pairs of the form $\Gamma \times (Q \cup \{\mathfrak{l}, \mathfrak{r}\})$. The additional component in the labels of these strong nodes indicates whether a position is left or right to the position of the head (and contains the state of the Turing machine for the position of the head). So, the above configuration corresponds to the sequence of strong nodes labeled $(a_1, \mathfrak{l}), \dots, (a_{n-1}, \mathfrak{l}), (a_n, q), (a_{n+1}, \mathfrak{r}), \dots, (a_{P(|w|)}, \mathfrak{r})$. We will denote these vertices as *cell nodes*.

Besides the cell nodes, we define a component containing $2 \cdot |\Delta|$ vertices; these nodes allow Destructor to choose the transition if M is in an existential state and to disconnect the network if an accepting configuration is reached. This component consists of $|\Delta|$ weak nodes $v_1, \dots, v_{|\Delta|}$, called *choice nodes*, $|\Delta| - 1$ strong nodes $v'_1, \dots, v'_{|\Delta|-1}$, called *delay nodes*, and a single strong node v_{target}, called the *target node*. Each of the $|\Delta|$ choice nodes corresponds to a transition t_i of the transition relation Δ. Intuitively, the existential choice of a transition (i.e., from an existential state) is propagated via the delay nodes to the choice nodes, each of which is labeled with a transition $t_i \in \Delta$. Each rule that relabels the cell nodes according to an existential computation step involving a transition t_i also applies a relabeling action on the choice node for t_i. Destructor is able to delete $|\Delta| - 1$ of these weak choice nodes before Constructor can applies

such a rule. So, Destructor chooses a transition t_i in an existential state by deleting all choice nodes except the one for t_i. We define relabeling rules such that the rejecting state is reached if Destructor chooses an invalid transition; this is the case if Destructor leaves a choice node deactivated whose transition cannot be applied. In order to give Destructor the opportunity to delete $|\Delta| - 1$ of the choice nodes, a chain of $|\Delta| - 1$ delay nodes is used. The delay nodes carries labels in $\{\#, \downarrow\}$; the symbol $\#$ is the default labeling, and the $\downarrow$-label is used to propagate the choice in an existential state down to the weak nodes with a delay of $|\Delta| - 1$ turns. The target node, labeled with $\bot$, is used to restore all of the (possibly deleted) choice nodes after the transition was selected. Also this is the vertex that will be isolated if the computation ends in an accepting state, so that the network becomes disconnected if M reaches q_{accept}.

The initial network for the construction is depicted in Figure 3.9. The initial network depends on the problem instance; for a given Turing machine M and an input word $w = w_1 \cdots w_n$ we label the cell node u_1 with (u_1, q_0), and each cell node u_i with $2 \leq i \leq n$ is labeled with $(w_i, \mathfrak{r})$. The cell nodes $u_{n+1}, \ldots, u_{P(n)}$ are labeled with $(\text{␣}, \mathfrak{r})$.

In the following rules we use "reset" as an abbreviation for

$$\bot \xrightarrow{\text{move}} t_1 \,;\, t_1 \xrightarrow{\text{move}} \bot \,; \ldots ; \bot \xrightarrow{\text{move}} t_n \,;\, t_n \xrightarrow{\text{move}} \bot$$

with $t_1, \ldots, t_n \in \Delta$. This reset-sequence is not a stand-alone rule, but is used as an ingredient for other rules whenever all choice nodes should be restored.

The rule set R is defined as follows. In order to simulate transitions in universal states, for each transition $(q, a, p, b, X) \in \Delta$ with $g(q) = \wedge$ and every $c \in \Gamma$ we add the rule

$$\big\langle (c, \mathfrak{l}), (a, q) \longrightarrow (c, p), (b, \mathfrak{r}) \,;\, \text{reset} \big\rangle$$

if $X = L$, and

$$\big\langle (a, q), (c, \mathfrak{r}) \longrightarrow (b, \mathfrak{l}), (c, p) \,;\, \text{reset} \big\rangle$$

if $X = R$. Towards the simulation of transitions in existential states we add the rules

$$\big\langle (c, q), \# \longrightarrow (c, q), \downarrow \big\rangle$$

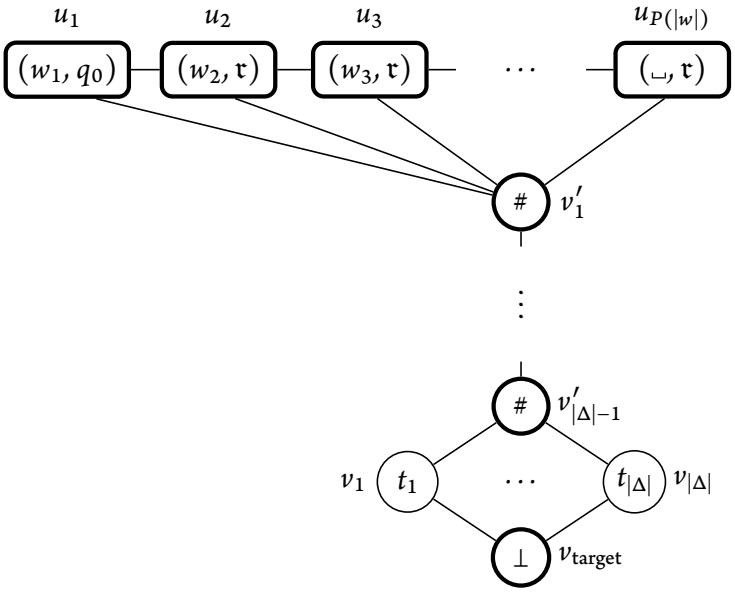

Figure 3.9: The initial network of a safety game representing the initial configuration of an alternating Turing machine.

for all $c \in \Gamma$ and $q \in Q$ with $g(q) = \vee$ and

$$\left\langle \downarrow, \# \longrightarrow \#, \downarrow \right\rangle$$

to propagate the $\downarrow$-label to the choice nodes when M is in an existential state. Then, for each transition $t_i = (q, a, p, b, X) \in \Delta$ with $g(q) = \vee$ and for all $c \in \Gamma$ we add rules

$$\left\langle \downarrow, t_i \longrightarrow \#, t_i \,;\, (c, \mathfrak{l}), (a, q) \longrightarrow (c, p), (b, \mathfrak{r}) \,;\, \text{reset} \right\rangle$$

if $X = L$, and

$$\left\langle \downarrow, t_i \longrightarrow \#, t_i \,;\, (a, q), (c, \mathfrak{r}) \longrightarrow (b, \mathfrak{l}), (c, p) \,;\, \text{reset} \right\rangle$$

if $X = R$. Note that the first part $\downarrow, t_i \longrightarrow \#, t_i$ of such a rule can only be applied if the choice node labeled with the transition $t_i = (q, a, p, b, X)$ has not been deleted after the last reset. In order to prevent Destructor from choosing a transition t_i that cannot be applied, we add rules

$$\left\langle \downarrow, t_i \longrightarrow \#, t_i \,;\, q', a' \longrightarrow q_{\text{reject}}, a' \,;\, \text{reset} \right\rangle$$

for all $a' \in \Gamma$, $q' \in Q$ with $q' \neq q$ or $a' \neq a$; these let Constructor proceed to the state q_{reject} immediately. Since we assumed that each configuration (except the accepting and rejecting configuration) has at least one successor configuration, Destructor will choose an applicable transition t_i. Finally, we add the rules

$$\left\langle q_{\text{reject}}, c \longrightarrow q_{\text{reject}}, c\,;\ \text{reset} \right\rangle$$

for arbitrary $c \in \Gamma$, which allow Constructor to keep the network connected permanently as soon as a rejecting configuration is reached.

To show the correctness of the construction, we first assume that M accepts w. Constructor continuously tries to simulate the computation of M in order to restore the choice node by applying a rule that contains the reset sequence; otherwise Destructor wins by deleting all of the $|\Delta|$ choice nodes. In a universal state, Constructor chooses a valid transition. In an existential state Destructor is able to delete $|\Delta| - 1$ choice nodes before Constructor is able to apply a transition; so, Destructor chooses a valid transition. Since M accepts w, Constructor finally generates a cell node that contains a q_{accept} in its label. This prevents Constructor from applying rules. Hence, Destructor wins in the next $|\Delta|$ turns by deleting all of the choice nodes (so, v_{target} becomes isolated).

Conversely, let us assume that M does not accept w. Then, M either reaches q_{reject} or never halts. In the first case Constructor preserves the network by simulating transitions of M forever. In the second case Constructor reaches a network containing a q_{reject} in its label, which allows her to restore the choice nodes in every turn. If Destructor misbehaves by choosing an invalid transition, Constructor is also able to reach a network that contains a q_{reject} label. Hence, in any case Constructor can preserve the connectivity of the network. □

3.3 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 3.3.1). Then we discuss decidable subcases. For solving these reachability games, we get an ExpTime upper

bound and a PSPACE lower bound (Section 3.3.2). For the unlabeled non-expanding case, we improve these results to a PSPACE upper bound and an NP lower bound (Section 3.3.3). At the end of this section we analyze non-expanding reachability games with multi-rules (Section 3.3.4).

3.3.1 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 3.2 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 3.11. *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

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

if $X = L$, and

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

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:

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

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

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

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

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

if $X = L$, and

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

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

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

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

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

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. □

3.3.2 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 3.12. In the non-expanding case, solving reachability connectivity games is in ExpTime.

Proof. We transform $\mathcal{G}$ into an *infinite two-player game* (see Thomas, 2008a; Grädel, Thomas, and Wilke, 2002) 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 Thomas, 1995, 2008a; Grädel, Thomas, and Wilke, 2002). □

Complementary to this ExpTime upper bound, we provide a Pspace lower bound.

Theorem 3.13. *In the non-expanding case, solving reachability connectivity games is Pspace-hard.*

Proof. We use a polynomial-time reduction from solving reachability sabotage games, which is Pspace-hard (see Theorem 1.4), to the problem of solving non-expanding reachability connectivity games. For every sabotage game $\mathcal{G}_s = (G_s, v_{\text{in}})$ on a game graph $G_s = (V_s, E_s)$ with a designated set $F \subseteq V_s$ of final vertices, 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 reachability 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 3.10 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

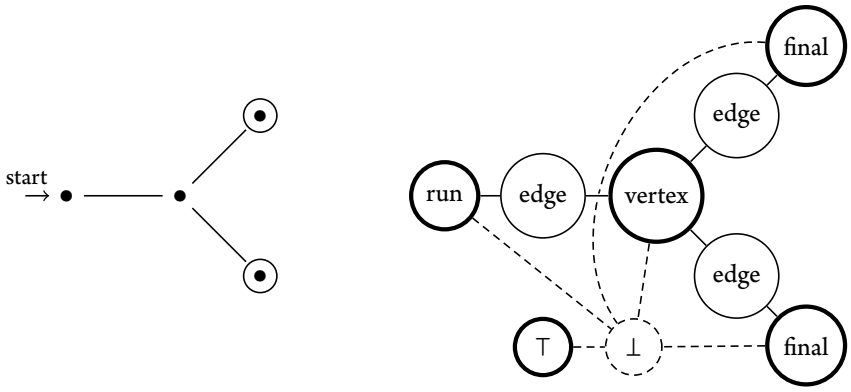

Figure 3.10: A game graph G_s of a sabotage game and its corresponding initial network G of a reachability game.

a second rule that additionally relabels a node from "final" to "reach" if it is reached by the "run" label:

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

The third rule $\left\langle \text{reach} \xrightarrow{\text{move}} \bot \right\rangle$ 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 reachability 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 3.7, where we reduce the state space by ignoring the deactivated nodes.

Theorem 3.14. *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 3.7 and Remark 3.12).

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 given game graph. □

Also for this fragment we obtain a Pspace lower bound.

Theorem 3.15. *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 3.13. In this construction we have to replace the movement rule $\left\langle \text{reach} \xrightarrow{\text{move}} \bot \right\rangle$,

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

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

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

3.3.3 Unlabeled Non-Expanding Games

In the unlabeled non-expanding case, where Constructor can only move strong nodes, we give an NP lower bound and a Pspace upper bound for solving reachability games.

Theorem 3.16. *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 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 $\left\langle \sqcup \xrightarrow{\text{move}} \sqcup \right\rangle$. 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 3.11 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

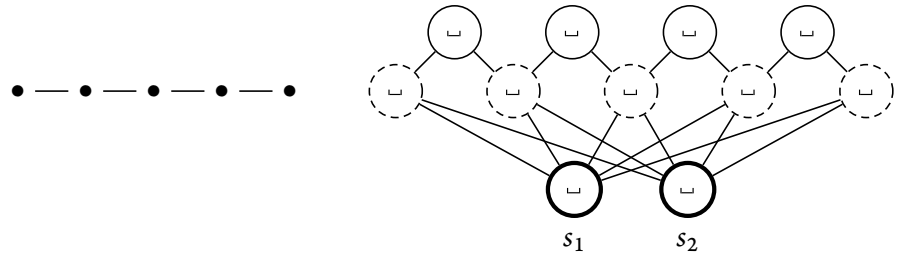

Figure 3.11: A graph G_{VC} and the corresponding initial network G of a reachability game for testing G_{VC} for a vertex cover of size two.

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 solving reachability games in 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 3.17. 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 3.18. *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 $u \in V \setminus \{v\}$ in both cases, 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 3.4 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 3.19. *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 3.18 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 3.6).

Theorem 3.20. *In the unlabeled non-expanding case, solving reachability connectivity games is decidable in Pspace.*

Proof. The proof is analogous to the proof of Theorem 3.6; we use the bound from Lemma 3.19 to truncate the game tree. □

3.3.4 Non-Expanding Multi-Rule Games

We have seen that we can solve non-expanding reachability games in ExpTime, but we only proved a Pspace lower bound. It is an open problem to close this gap. However, in the following we will prove that solving these non-expanding reachability games is ExpTime-complete if we also allow multi-rules.

Since the unfolding of a non-expanding multi-rule game is still at most exponential in the size of the given game, we can solve non-expanding multi-rule games in ExpTime as mentioned before (see Remark 3.12).

Remark 3.21. In the case of non-expanding multi-rule games, solving reachability connectivity games is in ExpTime.

We obtain the ExpTime-hardness by a reduction from the halting problem of polynomial space-bounded alternating Turing machines as we used it at the end of Section 3.2 to prove Theorem 3.10.

Theorem 3.22. *In the case of non-expanding multi-rule games, solving reachability connectivity games is ExpTime-hard.*

Proof. We reduce the accepting problem of polynomial space-bounded alternating Turing machines to the problem of solving non-expanding multi-rule reachability games. As in the proof of Theorem 3.10 we present alternating Turing machines in the format

$$M = (Q, \Gamma, \Delta, q_0, g)$$

with a state set Q, tape alphabet Γ (containing a blank symbol $\sqcup$), transition relation $\Delta \subseteq Q \times \Gamma \times Q \times \Gamma \times \{L, R\}$, initial state q_0, and a function $g\colon Q \to \{\wedge, \vee, \text{accept}, \text{reject}\}$ specifying the type of each state. We assume w.l.o.g. that q_{accept} (q_{reject}) is the only accepting (rejecting) state. We also assume w.l.o.g. that each configuration, except the accepting and rejecting configuration, has at least one successor configuration.

For a polynomial space-bounded alternating Turing machine M and an input word $w \in \Gamma^*$ we construct a game $\mathcal{G} = (G, R)$ such that M accepts w iff Constructor can preserve the connectivity of the network. As in the proof of Theorem 3.10 we represent the current configuration of the space-bounded Turing machine by a sequence of nodes. Constructor chooses the transitions in the existential states; Destructor chooses the transitions in the universal states (thus, in contrast to the proof of Theorem 3.10, Constructor is now the existential player). More precisely, we represent a Turing machine configuration $a_1 \dots a_{n-1} q a_n a_{n+1} \dots a_{P(|w|)}$ (where the head is on the a_n-labeled cell) by a sequence of $P(|w|)$ strong nodes labeled with these symbols in succession (for some polynomial P). We will denote these vertices as *cell nodes*. An additional component in the labels of these

strong nodes indicates whether a position is left or right to the position of the head (and contains the state of the Turing machine for the position of the head).

Besides the cell nodes, the network contains $2{\cdot}|\Delta|{+}2$ vertices. These are $|\Delta|$ weak nodes, called *choice nodes*, $|\Delta|-1$ strong nodes, called *delay nodes*, and three so-called *target nodes*. Each of the $|\Delta|$ choice nodes corresponds to a transition t_i of the transition relation Δ. The universal choice of a transition (i.e., from a universal state) is propagated via the delay nodes to the choice nodes. Each rule that relabels the cell nodes according to a universal computation step involving a transition t_i also applies a relabeling action on the choice node for t_i. Destructor is able to delete $|\Delta|-1$ of these weak choice nodes before Constructor applies such a rule. So, Destructor chooses a transition t_i in a universal state by deleting all choice nodes except the one for t_i. We define relabeling rules such that the accepting state is reached if Destructor chooses an invalid transition; that is the case if Destructor leaves a choice node deactivated whose transition cannot be applied. In order to give Destructor the opportunity to delete $|\Delta|-1$ of the choice nodes, a chain of $|\Delta|-1$ delay nodes is used (to delay Constructor's relabeling action involving the choice node). The three target nodes are labeled with $\top$, $+$, and $\bot$. The $\top$- and $\bot$-labeled nodes are strong, the $+$-labeled node is deactivated. The $\top$-labeled node is used to restore all of the (possibly deleted) choice nodes after the transition was selected. The $\bot$-labeled node is isolated in the graph of active nodes, but Constructor can establish a connection by restoring the $+$-labeled node (if she can simulate an accepting run of the Turing machine).

The initial network for the construction is depicted in Figure 3.12. The initial network depends on the problem instance; for a given Turing machine M and an input word $w = w_1 \cdots w_n$ we label the cell node u_1 with (u_1, q_0), and each cell node u_i with $2 \le i \le n$ is labeled with $(w_i, \mathfrak{r})$. The cell nodes $u_{n+1}, \ldots, u_{P(n)}$ are labeled with $(\sqcup, \mathfrak{r})$.

Analogously to the proof of Theorem 3.10, we use in Constructor's rules "reset" as an abbreviation for

$$\top \xrightarrow{\text{move}} t_1\,;\; t_1 \xrightarrow{\text{move}} \top\,;\; \ldots\,;\; \top \xrightarrow{\text{move}} t_n\,;\; t_n \xrightarrow{\text{move}} \top$$

with $t_1, \ldots, t_n \in \Delta$. The rules are defined as follows. In order to simulate transitions in existential states, for each transition $(q, a, p, b, X) \in \Delta$ with

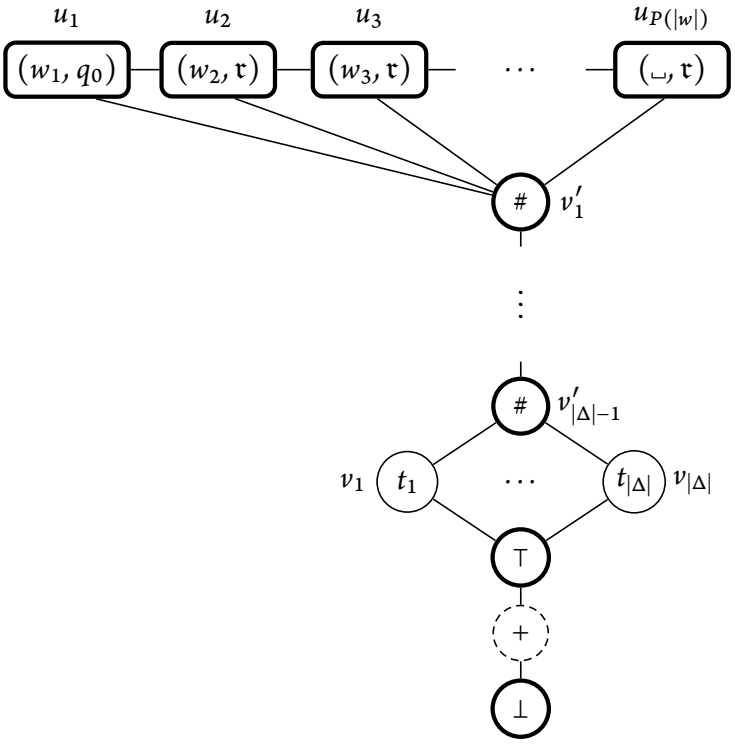

Figure 3.12: The initial network of a reachability game representing the initial configuration of an alternating Turing machine.

$g(q) = \vee$ and every $c \in \Gamma$ we add the rule

$$\left\langle (c, \mathfrak{l}), (a, q) \longrightarrow (c, p), (b, \mathfrak{r}) \,;\, \text{reset} \right\rangle$$

if $X = L$, and

$$\left\langle (a, q), (c, \mathfrak{r}) \longrightarrow (b, \mathfrak{l}), (c, p) \,;\, \text{reset} \right\rangle$$

if $X = R$. Towards the simulation of transitions in universal states we add rules

$$\left\langle (c, q), \# \longrightarrow (c, q), \downarrow \right\rangle$$

for all $c \in \Gamma$ and $q \in Q$ with $g(q) = \wedge$ and

$$\left\langle \downarrow, \# \longrightarrow \#, \downarrow \right\rangle$$

to propagate the ↓-label to the choice nodes when M is in a universal state. Then, for each transition $t_i = (q, a, p, b, X) \in \Delta$ with $g(q) = \wedge$ and for all $c \in \Gamma$ we add rules

$$\left\langle \downarrow, t_i \longrightarrow \#, t_i \,;\, (c, \mathfrak{l}), (a, q) \longrightarrow (c, p), (b, \mathfrak{r}) \,;\, \text{reset} \right\rangle$$

if $X = L$, and

$$\left\langle \downarrow, t_i \longrightarrow \#, t_i \,;\, (a, q), (c, \mathfrak{r}) \longrightarrow (b, \mathfrak{l}), (c, p) \,;\, \text{reset} \right\rangle$$

if $X = R$. In order to prevent Destructor from choosing a transition t_i that cannot be applied, we add rules

$$\left\langle \downarrow, t_i \longrightarrow \#, t_i \,;\, q', a' \longrightarrow q_{\text{accept}}, a' \,;\, \text{reset} \right\rangle$$

for all $a' \in \Gamma$, $q' \in Q$ with $q' \neq q$ or $a' \neq a$; these let Constructor proceed to the state q_{accept} immediately. Since we assumed that each configuration (except the accepting and rejecting configuration) has at least one successor configuration, Destructor will choose an applicable transition t_i. Finally, we add the rules

$$\left\langle q_{\text{accept}}, c \longrightarrow q_{\text{accept}}, c \,;\, \bot \xrightarrow{\text{move}} + \right\rangle$$

for arbitrary $c \in \Gamma$, which allow Constructor to restore the +-labeled node if an accepting configuration is reached.

To show the correctness of the construction, we first assume that M accepts w. Constructor simulates the computation of M to obtain a labeling of the cell nodes that represents an accepting configuration. As M accepts w, Constructor established a connected network by restoring the +-labeled node with the last-mentioned rule.

Conversely, if M does not accept w, Destructor can prevent Constructor from simulating an accepting computation of M (by accordant deletions of the choice nodes). Then, Constructor cannot restore the +-labeled node. Hence, the network always stays disconnected. □

3.4 Conclusion and Open Problems

In this chapter we introduced dynamic network connectivity games and studied the complexity of solving these games in both the reachability and

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
same but with multi-rules	ExpTime-complete	ExpTime-complete
unlabeled non-expanding	NP-hard / in Pspace	Pspace-complete

Table 3.1: 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. Multi-rules are only allowed for relabeling actions unless states otherwise (*with multi-rules*).

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 unlabeled fragment, where nodes are labeled uniformly, and the extension to multi-rule games, where single rules may contain multiple relabeling and movement actions. An overview of the results is given in Table 3.1.

The careful reader may have noticed that we have not discussed every possible restriction of the rule set in this chapter. Some of these missing cases are easy to analyze or 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 3.1. 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?

The results in this chapter 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 3.1 tight complexity bounds are missing.

Problem 3.2. 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 3.1?

Some of our results depend on the balance between node deletion and restoration; if Constructor restores a node, Destructor can delete another one immediately. For non-expanding games we showed that multi-rules increase the computational complexity. What happens in the expanding case where the solvability depends on this balance?

Problem 3.3. Are reachability connectivity games with multi-rules algorithmically solvable if only weak node creation and relabelings are allowed? Are safety connectivity games with multi-rules algorithmically solvable if weak node creation is forbidden? Are safety connectivity games with multi-rules algorithmically solvable if movement rules are forbidden?

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 (see Grädel, Thomas, and Wilke, 2002). We expect at least that the negative results in this chapter (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 3.4. To what extent can the results in this chapter 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 (Grüner, 2011). The upper bound is obtained by reducing these games to LTL games on graphs, whose solvability has been shown to be complete for 2ExpTime (Pnueli and Rosner, 1989; also see Rosner, 1991; Alur, La Torre, and Madhusudan, 2003). The lower bound is shown by a reduction from polynomial space-bounded alternating Turing machines; this proof is a variant of our proofs for the ExpTime lower bound for solving non-expanding multi-rule games (see Theorems 3.10 and 3.22). However, Grüner's proof does not use multi-rules; instead 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, which could be used to overcome some of the simplifications

that are built into our connectivity game model. One of these simplifications is the fact that the player Destructor acts as an omniscient adversary. In the view of verifying fault-tolerant systems, however, such an omniscient adversary who deletes nodes is rarely realistic; faults are better modeled as random events. We studied this scenario in the framework of sabotage games (Chapter 1). One can study the corresponding case for connectivity games, where each action of Destructor is replaced by a random vertex deletion. It is easy to adapt the hardness results at least for non-expanding games: Grüner (2011) showed that sabotage games can be simulated by randomized connectivity games; in these games Constructor wins with a parametrized probability that can be analyzed in the same way as the probability $p_{k,n}$ in Sections 1.5 and 1.6. Thus, solving non-expanding reachability and safety connectivity games in the randomized variant is PSPACE-hard even if the probability we ask for is restricted to an arbitrary small interval (as in Section 1.5). To obtain a lower bound for solving the randomized version of non-expanding reachability and safety games, Grüner (2011) proposed to unfold the connectivity game resulting in a Markov decision process (see Filar and Vrieze, 1996; Puterman, 2005; Baier and Katoen, 2008). The size of the unfolding is at most exponential in the size of the given game and the maximal and the minimal reachability probability in a Markov decision process can be computed in polynomial time. Thus, both reachability and safety connectivity games are solvable in EXPTIME.

Another simplification of our model is that Destructor and Constructor have complete information about the network and its current state. One might pursue a model where the Constructor has only partial information of the network or – more challenging – a model where each strong node is an autonomous player, each of which with its own information set. Regarding the first case, deciding which player wins in a two-player game of incomplete information is known to be much harder (Reif, 1984). In the second case, where individual strong nodes build a coalition, we yield a multiplayer game of imperfect information. In general, however, determining the winner for such games is undecidable (Azhar, Peterson, and Reif, 2001).

Indeed, the game model in this chapter 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

chapter) 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 combine the connectivity game model with the routing game model from 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 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 by Grüner (2011). These yield small (although not optimal) solutions with efficient winning strategies on various classes of networks.

Bibliography

Aho, Alfred V. (1990). Algorithms for Finding Patterns in Strings. In: *Handbook of Theoretical Computer Science, Volume A: Algorithms and Complexity*, pp 255–300. Elsevier and MIT Press.

Albers, Susanne (2003). Online algorithms: a survey. In: *Mathematical Programming*, vol. 97(1–2), pp 3–26. doi: 10.1007/s10107-003-0436-0.

Altman, Eitan, Thomas Boulogne, Rachid El-Azouzi, Tania Jiménez, and Laura Wynter (2006). A survey on networking games in telecommunications. In: *Computers & Operations Research*, vol. 33, pp 286–311. doi: 10.1016/j.cor.2004.06.005.

Alur, Rajeev, Salvatore La Torre, and P. Madhusudan (2003). Playing Games with Boxes and Diamonds. In: *Proceedings of the 14th International Conference on Concurrency Theory, CONCUR 2003*. Vol. 2761 of Lecture Notes in Computer Science, pp 127–141. Springer. doi: 10.1007/978-3-540-45187-7_8.

Awerbuch, Baruch, Petra Berenbrink, André Brinkmann, and Christian Scheideler (2001). Simple Routing Strategies for Adversarial Systems. In: *Proceedings of the 42nd Annual Symposium on Foundations of Computer Science, FOCS 2001*, pp 158–167. IEEE Computer Society Press. doi: 10.1109/SFCS.2001.959890.

Awerbuch, Baruch, André Brinkmann, and Christian Scheideler (2003). Anycasting in Adversarial Systems: Routing and Admission Control. In: *Proceedings of the 30th International Colloquium on Automata, Languages and Programming, ICALP 2003*. Vol. 2719 of Lecture Notes in Computer Science, pp 1153–1168. Springer. doi: 10.1007/3-540-45061-0_88.

Awerbuch, Baruch, Yishay Mansour, and Nir Shavit (1989). Polynomial End-To-End Communication (Extended Abstract). In: *Proceedings of the 30th*

Annual Symposium on Foundations of Computer Science, FOCS 1989, pp 358–363. IEEE Computer Society Press. DOI: 10.1109/SFCS.1989.63503.

Azhar, Salman, Gary L. Peterson, and John H. Reif (2001). Lower Bounds for Multiplayer Non-Cooperative Games of Incomplete Information. In: *Journal of Computers and Mathematics with Applications*, vol. 41(7–8), pp 957–992. DOI: 10.1016/S0898-1221(00)00333-3.

Baier, Christel and Joost-Pieter Katoen (2008). *Principles of Model Checking*. MIT Press.

van Benthem, Johan (2005). An Essay on Sabotage and Obstruction. In: *Mechanizing Mathematical Reasoning: Essays in Honor of Jörg H. Siekmann on the Occasion of His 60th Birthday*. Vol. 2605 of Lecture Notes in Computer Science, pp 268–276. Springer. DOI: 10.1007/978-3-540-32254-2_16.

Bloem, Roderick, Stefan J. Galler, Barbara Jobstmann, Nir Piterman, Amir Pnueli, and Martin Weiglhofer (2007). Automatic Hardware Synthesis from Specifications: A Case Study. In: *Proceedings of the 2007 Design, Automation and Test in Europe Conference and Exposition, DATE 2007*, pp 1188–1193. ACM Press. DOI: 10.1145/1266366.1266622.

Borodin, Allan, Jon M. Kleinberg, Prabhakar Raghavan, Madhu Sudan, and David P. Williamson (1996). Adversarial Queueing Theory. In: *Proceedings of the 28th Annual ACM Symposium on the Theory of Computing, STOC 1996*, pp 376–385. ACM Press. DOI: 10.1145/237814.237984.

Borodin, Allan and Ran El-Yaniv (1998). *Online Computation and Competitive Analysis*. Cambridge University Press.

Büchi, J. Richard and Lawrence H. Landweber (1969). Solving Sequential Conditions by Finite-State Strategies. In: *Transactions of the American Mathematical Society*, vol. 138, pp 295–311. DOI: 10.2307/1995086.

Casteigts, Arnaud, Paola Flocchini, Bernard Mans, and Nicola Santoro (2010). Deterministic Computations in Time-Varying Graphs: Broadcasting under Unstructured Mobility. In: *Proceedings of the 6th IFIP International Conference on Theoretical Computer Science, IFIP TCS 2010*. Vol. 323 of IFIP Advances in Information and Communication Technology, pp 111–124. Springer. DOI: 10.1007/978-3-642-15240-5_9.

Casteigts, Arnaud, Paola Flocchini, Walter Quattrociocchi, and Nicola Santoro (2011). Time-Varying Graphs and Dynamic Networks. In: *Proceedings of the 10th International Conference on Ad-Hoc Networks and Wireless,*

ADHOC-NOW 2011. Vol. 6811 of Lecture Notes in Computer Science, pp 346–359. Springer. DOI: 10.1007/978-3-642-22450-8_27.

CHANDRA, Ashok K., Dexter KOZEN, and Larry J. STOCKMEYER (1981). Alternation. In: *Journal of the ACM*, vol. 28(1), pp 114–133. DOI: 10.1145/322234.322243.

CHATTERJEE, Krishnendu and Thomas A. HENZINGER (2012). A survey of stochastic ω-regular games. In: *Journal of Computer and System Sciences*, vol. 78(2), pp 394–413. DOI: 10.1016/j.jcss.2011.05.002.

CHURCH, Alonzo (1957). Applications of recursive arithmetic to the problem of cricuit synthesis. In: *Summaries of the Summer Institute of Symbolic Logic*, vol. I, pp 3–50. Cornell University, Ithaca, NY, USA.

——— (1963). Logic, Arithmetic and Automata. In: *Proceedings of the International Congress of Mathematicians 1962*, pp 23–35. Institut Mittag-Leffler, Djursholm, Sweden.

CLARKE Jr., Edmund M., Orna GRUMBERG, and Doron A. PELED (2000). *Model Checking*. MIT Press.

COSTA, Marie-Christine, Lucas LÉTOCART, and Frédéric ROUPIN (2005). Minimal multicut and maximal integer multiflow: A survey. In: *European Journal of Operational Research*, vol. 162(1), pp 55–69. DOI: 10.1016/j.ejor.2003.10.037.

COURCOUBETIS, Costas and Mihalis YANNAKAKIS (1995). The Complexity of Probabilistic Verification. In: *Journal of the ACM*, vol. 42(4), pp 857–907. DOI: 10.1145/210332.210339.

DEMETRESCU, Camil, David EPPSTEIN, Zvi GALIL, and Giuseppe F. ITALIANO (2010). Dynamic graph algorithms. In: *Algorithms and Theory of Computation Handbook, Second Edition, Volume 1: General Concepts and Techniques*. Chap. 9, pp 9-1–9-28. CRC Press. DOI: 10.1201/9781584888239-c9.

DEMRI, Stéphane (2005). A reduction from DLP to PDL. in: *Journal of Logic and Computation*, vol. 15(5), pp 767–785. DOI: 10.1093/logcom/exi043.

EVEN, Shimon, Alon ITAI, and Adi SHAMIR (1976). On the Complexity of Timetable and Multicommodity Flow Problems. In: *SIAM Journal on Computing*, vol. 5(4), pp 691–703. DOI: 10.1137/0205048.

FEIGENBAUM, Joan and Sampath KANNAN (2000). Dynamic Graph Algorithms. In: *Handbook of Discrete and Combinatorial Mathematics*. Chap. 17.5, pp 1142–1151. CRC Press. DOI: 10.1201/9781439832905.ch17.

Fiat, Amos and Gerhard J. Woeginger, eds. (1998). *Online Algorithms: The State of the Art*. Vol. 1442 of Lecture Notes in Computer Science. Springer. doi: 10.1007/BFb0029561.

Filar, Jerzy A. and Koos Vrieze (1996). *Competitive Markov Decision Processes*. Springer.

Gabbay, Dov M. (2008). Introducing Reactive Kripke Semantics and Arc Accessibility. In: *Pillars of Computer Science: Essays Dedicated to Boris (Boaz) Trakhtenbrot on the Occasion of His 85th Birthday*. Vol. 4800 of Lecture Notes in Computer Science, pp 292–341. Springer. doi: 10.1007/978-3-540-78127-1_17.

Gadducci, Fabio, Reiko Heckel, and Manuel Koch (1998). A Fully Abstract Model for Graph-Interpreted Temporal Logic. In: *Proceedings of the 6th International Workshop on Theory and Application of Graph Transformations, TAGT 1998*. Vol. 1764 of Lecture Notes in Computer Science, pp 310–322. Springer. doi: 10.1007/978-3-540-46464-8_22.

Gale, David and Frank M. Stewart (1953). Infinite Games with Perfect Information. In: *Contributions to the Theory of Games, Volume II*. Annals of Mathematics Studies 28, pp 245–266. Princeton University Press.

Garg, Naveen and Jochen Könemann (2007). Faster and Simpler Algorithms for Multicommodity Flow and Other Fractional Packing Problems. In: *SIAM Journal on Computing*, vol. 37(2), pp 630–652.

von zur Gathen, Joachim and Jürgen Gerhard (2003). *Modern Computer Algebra*. Second Edition. Cambridge University Press.

Gierasimczuk, Nina, Lena Kurzen, and Fernando R. Velázquez-Quesada (2009). Learning and Teaching as a Game: A Sabotage Approach. In: *Proceedings of the Second International Workshop on Logic, Rationality, and Interaction, LORI 2009*. Vol. 5834 of Lecture Notes in Computer Science, pp 119–132. Springer. doi: 10.1007/978-3-642-04893-7_10.

Göller, Stefan and Markus Lohrey (2006). Infinite State Model-Checking of Propositional Dynamic Logics. In: *Proceedings of the 20th International Workshop on Computer Science Logic, CSL 2006*. Vol. 4207 of Lecture Notes in Computer Science, pp 349–364. Springer. doi: 10.1007/11874683_23.

Grädel, Erich, Wolfgang Thomas, and Thomas Wilke, eds. (2002). *Automata, Logics, and Infinite Games: A Guide to Current Research*. Vol. 2500 of Lecture Notes in Computer Science. Springer. doi: 10.1007/3-540-36387-4.

Gross, James, Frank G. Radmacher, and Wolfgang Thomas (2010). A Game-Theoretic Approach to Routing under Adversarial Conditions. In: *Proceed-*

ings of the 6th IFIP International Conference on Theoretical Computer Science, IFIP TCS 2010. Vol. 323 of IFIP Advances in Information and Communication Technology, pp 355–370. Springer. DOI: 10.1007/978-3-642-15240-5_26.

GRÜNER, Sten (2011). Game Theoretic Analysis of Dynamic Networks. Diploma thesis. RWTH Aachen, Germany.

GRÜNER, Sten, Frank G. RADMACHER, and Wolfgang THOMAS (2011). Connectivity Games over Dynamic Networks. In: *Proceedings of the Second International Symposium on Games, Automata, Logics and Formal Verification, GandALF 2011*. Vol. 54 of Electronic Proceedings in Theoretical Computer Science, pp 131–145. DOI: 10.4204/EPTCS.54.10.

——— (2012). Connectivity Games over Dynamic Networks. In: *Theoretical Computer Science*. To appear. DOI: 10.1016/j.tcs.2012.12.001.

HANSSON, Hans and Bengt JONSSON (1994). A Logic for Reasoning about Time and Reliability. In: *Formal Aspects of Computing*, vol. 6(5), pp 512–535. DOI: 10.1007/BF01211866.

HECKEL, Reiko (2006). Graph Transformation in a Nutshell. In: *Proceedings of the School of SegraVis Research Training Network on Foundations of Visual Modelling Techniques, FoVMT 2004*. Vol. 148(1) of Electronic Notes in Theoretical Computer Science, pp 187–198. Elsevier. DOI: 10.1016/j.entcs.2005.12.018.

HOLM, Jacob, Kristian DE LICHTENBERG, and Mikkel THORUP (2001). Poly-logarithmic deterministic fully-dynamic algorithms for connectivity, minimum spanning tree, 2-edge, and biconnectivity. In: *Journal of the ACM*, vol. 48(4), pp 723–760. DOI: 10.1145/502090.502095.

KECHRIS, Alexander S. (1995). *Classical Descriptive Set Theory*. Vol. 156 of Graduate Texts in Mathematics. Springer.

KLEIN, Dominik (2008). Solving Randomized Sabotage Games for Navigation in Networks. Diploma thesis. RWTH Aachen, Germany.

KLEIN, Dominik, Frank G. RADMACHER, and Wolfgang THOMAS (2009). The Complexity of Reachability in Randomized Sabotage Games. In: *Proceedings of the 3rd IPM International Conference on Fundamentals of Software Engineering, FSEN 2009*. Vol. 5961 of Lecture Notes in Computer Science, pp 162–177. Springer. DOI: 10.1007/978-3-642-11623-0_9.

——— (2012). Moving in a Network under Random Failures: A Complexity Analysis. In: *Science of Computer Programming*, vol. 77(7–8), pp 940–954. DOI: 10.1016/j.scico.2010.05.009.

Kupferman, Orna, Moshe Y. Vardi, and Pierre Wolper (2001). Module Checking. In: *Information and Computation*, vol. 164(2), pp 322–344.

Kurzen, Lena (2011). Complexity in Interaction. PhD thesis. Institute for Logic, Language and Computation, Amsterdam, Netherlands. URL: http://dare.uva.nl/de/record/399235.

Littman, Michael L., Stephen M. Majercik, and Toniann Pitassi (2001). Stochastic Boolean Satisfiability. In: *Journal of Automated Reasoning*, vol. 27(3), pp 251–296.

Löding, Christof and Philipp Rohde (2003a). *Solving the Sabotage Game is PSPACE-hard*. Tech. rep. AIB-05-2003. RWTH Aachen, Germany. URL: http://aib.informatik.rwth-aachen.de/2003/.

——— (2003b). Solving the Sabotage Game is PSPACE-hard. In: *Proceedings of the 28th International Symposium on Mathematical Foundations of Computer Science, MFCS 2003*. Vol. 2747 of Lecture Notes in Computer Science, pp 531–540. Springer. DOI: 10.1007/978-3-540-45138-9_47.

——— (2003c). Model Checking and Satisfiability for Sabotage Modal Logic. In: *Proceedings of the 23rd Conference on Foundations of Software Technology and Theoretical Computer Science, FSTTCS 2003*. Vol. 2914 of Lecture Notes in Computer Science, pp 302–313. Springer. DOI: 10.1007/978-3-540-24597-1_26.

Martin, Donald A. (1975). Borel determinacy. In: *Annals of Mathematics*, vol. 102(2), pp 363–371. DOI: 10.2307/1971035.

——— (1985). A Purely Inductive Proof of Borel Determinacy. In: *Recursion Theory*. Vol. 42 of Proceedings of Symposia in Pure Mathematics, pp 303–308. American Mathematical Society.

Papadimitriou, Christos H. (1985). Games Against Nature. In: *Journal of Computer and System Sciences*, vol. 31(2), pp 288–301. DOI: 10.1016/0022-0000(85)90045-5.

——— (1994). *Computational Complexity*. Addison Wesley.

Pnueli, Amir and Roni Rosner (1989). On the Synthesis of an Asynchronous Reactive Module. In: *Proceedings of the 16th International Colloquium on Automata, Languages and Programming, ICALP 1989*. Vol. 372 of Lecture Notes in Computer Science, pp 652–671. Springer. DOI: 10.1007/BFb0035790.

Pucella, Riccardo and Vicky Weissman (2004). Reasoning about Dynamic Policies. In: *Proceedings of the 7th International Conference on Foundations of Software Science and Computation Structures, FoSSaCS 2004*. Vol. 2987 of

Lecture Notes in Computer Science, pp 453–467. Springer. DOI: 10.1007/978-3-540-24727-2_32.

PUTERMAN, Martin L. (2005). *Markov Decision Processes: Discrete Stochastic Dynamic Programming*. Wiley Series in Probability and Statistics. Wiley-Interscience.

RABIN, Michael O. (1969). Decidability of Second-Order Theories and Automata on Infinite Trees. In: *Transactions of the American Mathematical Society*, vol. 141, pp 1–35. DOI: 10.2307/1994916.

RADMACHER, Frank G. and Wolfgang THOMAS (2008). A Game Theoretic Approach to the Analysis of Dynamic Networks. In: *Proceedings of the 1st Workshop on Formal Verification of Adaptive Systems, VerAS 2007*. Vol. 200(2) of Electronic Notes in Theoretical Computer Science, pp 21–37. Elsevier. DOI: 10.1016/j.entcs.2008.02.010.

RAJARAMAN, Rajmohan (2002). Topology control and routing in ad hoc networks: a survey. In: *SIGACT News*, vol. 33(2), pp 60–73. DOI: 10.1145/564585.564602.

REIF, John H. (1984). The Complexity of Two-Player Games of Incomplete Information. In: *Journal of Computer and System Sciences*, vol. 29(2), pp 274–301. DOI: 10.1016/0022-0000(84)90034-5.

RODITTY, Liam and Uri ZWICK (2004). A fully dynamic reachability algorithm for directed graphs with an almost linear update time. In: *Proceedings of the 36th Annual ACM Symposium on Theory of Computing, STOC 2004*, pp 184–191. ACM. DOI: 10.1145/1007352.1007387.

ROHDE, Philipp (2004). Moving in a Crumbling Network: The Balanced Case. In: *Proceedings of the 18th International Workshop on Computer Science Logic, CSL 2004*. Vol. 3210 of Lecture Notes in Computer Science, pp 310–324. Springer. DOI: 10.1007/978-3-540-30124-0_25.

——— (2005). On Games and Logics over Dynamically Changing Structures. PhD thesis. RWTH Aachen, Germany. URL: http://darwin.bth.rwth-aachen.de/opus3/volltexte/2006/1380/.

ROSNER, Roni (1991). Modular Synthesis of Reactive Systems. PhD thesis. The Weizmann Institute of Science, Rehovot, Israel.

ROZENBERG, Grzegorz, ed. (1997). *Handbook of Graph Grammars and Computing by Graph Transformations, Volume 1: Foundations*. World Scientific.

SCHEIDELER, Christian (2002). Models and Techniques for Communication in Dynamic Networks. In: *Proceedings of the 19th Annual Symposium on Theoret-*

ical Aspects of Computer Science, STACS 2002. Vol. 2285 of Lecture Notes in Computer Science, pp 27–49. Springer. DOI: 10.1007/3-540-45841-7_2.

Shahrokhi, Farhad and David W. Matula (1990). The Maximum Concurrent Flow Problem. In: *Journal of the ACM*, vol. 37(2), pp 318–334. DOI: 10.1145/77600.77620.

Thomas, Wolfgang (1995). On the Synthesis of Strategies in Infinite Games. In: *Proceedings of the 12th Annual Symposium on Theoretical Aspects of Computer Science, STACS 1995*. Vol. 900 of Lecture Notes in Computer Science, pp 1–13. Springer. DOI: 10.1007/3-540-59042-0_57.

——— (2008a). Solution of Church's problem: A tutorial. In: *New Perspectives on Games and Interaction*. Vol. 4 of Texts in Logic and Games, pp 211–236. Amsterdam University Press. DOI: 10.5117/9789089640574.

——— (2008b). Church's Problem and a Tour through Automata Theory. In: *Pillars of Computer Science*. Vol. 4800 of Lecture Notes in Computer Science, pp 635–655. Springer. DOI: 10.1007/978-3-540-78127-1_35.

——— (2009). Facets of Synthesis: Revisiting Church's Problem. In: *Proceedings of the 12th International Conference on Foundations of Software Science and Computational Structures, FoSSaCS 2009*. Vol. 5504 of Lecture Notes in Computer Science, pp 1–14. Springer. DOI: 10.1007/978-3-642-00596-1_1.

Vardi, Moshe Y. and Pierre Wolper (1994). Reasoning About Infinite Computations. In: *Information and Computation*, vol. 115(1), pp 1–37. DOI: 10.1006/inco.1994.1092.

Weber, Volker and Thomas Schwentick (2007). Dynamic Complexity Theory Revisited. In: *Theory of Computing Systems*, vol. 40(4), pp 355–377. DOI: 10.1007/s00224-006-1312-0.

Zielonka, Wiesław (1998). Infinite Games on Finitely Coloured Graphs with Applications to Automata on Infinite Trees. In: *Theoretical Computer Science*, vol. 200(1–2), pp 135–183. DOI: 10.1016/S0304-3975(98)00009-7.

Index