A Formal Proof for the Correctness of Tangle Learning

Master's Thesis in Computer Science

Suzanne Ellen van der Veen

April 5, 2024

Committee:
Dr. Tom van Dijk
Dr. Peter Lammich
Dr. ir. Pieter-Tjerk de Boer

Faculty of Electrical Engineering,
Mathematics and Computer Science
University of Twente
Enschede, The Netherlands

Acknowledgements

First of all, I would like to thank the members of my graduation committee Tom van Dijk, Peter Lammich, and Pieter-Tjerk de Boer for their feedback on the thesis. As my daily supervisors, I would further like to thank Tom and Peter for their advice and assistance during my project. Specifically, Peter's experience with Isabelle and proofs and Tom's extensive knowledge of parity games and tangle learning were vital in the project.

I would also like to thank Alexander Stekelenburg, who was working on his Master's thesis about parity games and tangle learning at the same time as myself. His advice for the thesis is greatly appreciated, and we have had many useful discussions about our thesis topics.

Finally, I would like to thank my parents, my brother, my partner Maple, and my friends for their continued support and advice during the project. Without them, I would not have been able to complete it.

Abstract

Parity games are a type of two-player game that have applications in model checking. Scientifically, they are interesting because their complexity indicates that they might be solved in polynomial time, but no algorithm capable of solving them in polynomial time has been discovered as of yet. Tangle learning is one algorithm that aims to solve parity games, which has a correctness proof on paper, but no computer-verified proof.

This thesis focuses on a correctness proof of tangle learning using the Isabelle/HOL interactive theorem prover, with the aim of improving confidence in the correctness of the algorithm and exploring how such proofs can be done. We first define basic concepts of parity games and prove important properties thereof. Then, we give definitions for concepts related to tangle learning. We define our tangle attractor algorithm and the `search` algorithm, one of two algorithms that together constitute tangle learning, using inductive predicates for step relations of loops. We then prove the correctness of our tangle attractor algorithm and finally prove that the `search` algorithm finds a finite set of tangles that is nonempty if its input region is nonempty, and that `search` terminates. We have not completed the correctness proof of `solve`, the second algorithm that is part of tangle learning.

Keywords: Parity games, Tangle learning, Interactive theorem proving, Isabelle/HOL

Contents

1 Introduction **1**
- 1.1 Research Goals 2
- 1.2 Overview 2

2 Background **3**
- 2.1 Parity Games 3
- 2.2 Interactive Theorem Proving 4

3 Preliminaries **5**
- 3.1 Parity Games 5
  - 3.1.1 Attractors 6
  - 3.1.2 Dominions 7
  - 3.1.3 Tangles 7
  - 3.1.4 Tangle Attractors 8
  - 3.1.5 Tangle Learning 8
- 3.2 Definitions and Proofs in Isabelle 9
  - 3.2.1 Theories 9
  - 3.2.2 Datatypes 10
  - 3.2.3 Lists 10
  - 3.2.4 Sets 10
  - 3.2.5 Nonrecursive Definitions 11
  - 3.2.6 Recursive Functions 11
  - 3.2.7 Inductive Definitions 12
  - 3.2.8 Contexts and Locales 12
  - 3.2.9 Lemmas, Theorems, and Corollaries 12
  - 3.2.10 Automated Proofs 13
  - 3.2.11 Isar Proofs 14

4 Formalisation of Parity Games **16**
- 4.1 Basic Concepts 16
  - 4.1.1 Directed Graphs 16
  - 4.1.2 Fixing a Finite Number of Nodes 17
  - 4.1.3 Graphs Without Dead Ends 17
  - 4.1.4 Strategies and Arenas 18
  - 4.1.5 Parity Games and Players 18
  - 4.1.6 Winning Regions 19
- 4.2 Plays 21
- 4.3 Attractors 21
- 4.4 Proof of Positional Determinancy 27

5 Formalisation of Tangle Learning **32**
- 5.1 Tangles 32
- 5.2 Tangle Attractors 34
  - 5.2.1 Correctness Proof 35
  - 5.2.2 Termination Proof 60
  - 5.2.3 Further Properties 61

5.2.4 α-Maximality . . . 63
5.3 The Search Algorithm . . . 64
5.3.1 Correctness Proof . . . 65
5.3.2 Termination Proof . . . 77
5.4 The Solve Algorithm . . . 80

6 Related Work **81**
6.1 Formalisation of Parity Games and Positional Determinancy . . . 81
6.2 Termination Proof of Zielonka's Recursive Algorithm . . . 83

7 Discussion **85**
7.1 Challenges . . . 85
7.2 Van Dijk's Correctness Lemmas . . . 86
7.3 Further Properties . . . 87
7.4 Validity of Our Definitions . . . 87
7.5 Design Decisions and Potential Improvements . . . 88

8 Conclusion **90**
8.1 Future Work . . . 91

Bibliography **92**

Chapter 1

Introduction

A parity game is played by two players, Even and Odd. Each of these players owns a set of nodes in a finite directed graph without dead ends. A game starts on some arbitrary node, from which the player who owns it decides which successor to move to. The owner of the destination then decides where to move. This is repeated infinitely to form a play. Every node in the graph has also been assigned a natural number, its priority. A play is won by Even if the highest priority encountered infinitely often is even; if it is odd, the play is won by Odd.

A prominent application of parity games is model checking. This is the practice of using models of software (or hardware) systems to verify their properties. One approach consists of converting a system and property expressed in the μ-calculus into a parity game, a transformation which can be performed in polynomial time [10, 11, 26]. The μ-calculus encompasses temporal logics such as CTL and CTL*, meaning that we can check properties expressed in these logics by solving a parity game. Conversely, we can use parity games to generate verified implementations in a process known as LTL synthesis [21]. A system specification in the form of an LTL property is transformed into a parity game, which is then solved. This solution is then converted into a model for a verified implementation.

Another reason parity games are scientifically interesting is their complexity. Jurdziński [16] showed that the problem of solving a parity game lies in **UP** $\cap$ **co-UP**, a subset of **NP** $\cap$ **co-NP**. It is a common conjecture in the field that problems in **NP** $\cap$ **co-NP** have a polynomial solution—though whether or not this is actually true remains an open question. Although Calude et al. [6] found a quasipolynomial solution in 2017, and various other authors [18, 20] have found their own quasipolynomial solutions since, a polynomial solution has yet to be found.

Tangle learning is an algorithm for solving parity games that was introduced by Van Dijk [7]. It relies on the concept of a tangle, informally described as a set of nodes in the game where one player has a strategy such that every node in the set can be reached from every other node in the set, and all plays that stay within this set of nodes are won by that player. The algorithm solves a game by iteratively decomposing the graph into tangles and extending them until they reach their maximal size.

Van Dijk provided a pen-and-paper proof for the correctness of tangle learning. Although we have reasonable confidence in the validity of this proof, and we can verify it ourselves, these kinds of proofs may contain unnoticed logical errors. As a result, there is always room for doubt about the validity of such a proof.

To solve this problem, we can use Isabelle, a proof assistant for interactive theorem proving [23]. Interactive theorem proving is a practice where software is used to aid in the verification of mathematical proofs. Isabelle allows the user to specify mathematical properties and programs in a Haskell-like language. It is equipped with various automated provers for proving relatively simple properties. More complex, involved proofs can also be verified step-by-step using a language called Isar. Isabelle itself is thoroughly verified and employs multiple fail-safes to ensure it never accepts a proof as valid unless this is true beyond any doubt. Therefore, proofs made and verified with Isabelle enjoy a much greater level of confidence in their validity than those written on paper.

In this thesis, we work on the formulation of a machine-checked formal proof for the correctness of the tangle learning algorithm using Isabelle/HOL. With this proof, we would greatly increase confidence in the correctness of the algorithm. Furthermore, we may explore ways such a proof can be designed in Isabelle/HOL. This may help in the development of future correctness proofs for algorithms

like tangle learning in Isabelle. In part, the proof is an attempt at adapting Van Dijk's existing proof to work in Isabelle, as it is a solid starting point, but alternative approaches are also investigated. The Isabelle code for our proofs is available at doi.org/10.5281/zenodo.10932291.

1.1 Research Goals

The main aim of this thesis is to work towards the formulation of a machine-checked formal proof for the correctness of Tangle Learning. To this end, it mainly attempts to answer the following research question:

- **How can the correctness of Tangle Learning be proven in Isabelle/HOL?**
  Van Dijk provided a pen-and-paper proof for the correctness of Tangle Learning in his paper [7, Section 4.5]. However, the confidence in its correctness could be greatly increased by completing a computer-checked formal proof. Furthermore, by working on such a proof, we may discover new insights into how algorithms like tangle learning can be proven in Isabelle/HOL. Thus, the central problem this thesis aims to solve is the lack of such a proof.

  This main question is supported by three subquestions:

  1. **What are the challenges in proving the correctness of Tangle Learning in Isabelle/HOL?**
     A formal proof may be difficult to complete, which means it is important to identify the aspects of proving the correctness of the algorithm that present difficulties. The lessons learned may aid future attempts to prove correctness of related algorithms in Isabelle/HOL.
  2. **How can Van Dijk's correctness proof for Tangle Learning be translated into Isabelle/HOL?**
     Van Dijk's original pen-and-paper proof may offer an approach to complete the machine-checked proof. Translating it is a good starting point for attempting a proof in Isabelle/-HOL.
  3. **Which alternative properties should be proven to aid in a correctness proof in Isabelle/HOL?**
     It is important to not singularly focus on Van Dijk's proof, as it may actually contain errors, or it may simply not be viable in Isabelle/HOL. Therefore, we must look into alternative properties that may help prove correctness.

1.2 Overview

The rest of this thesis is divided into 7 chapters. In Chapter 2, we will discuss background on parity games and interactive theorem proving. Chapter 3 gives all necessary definitions related to parity games and tangle learning, as well as a short overview of relevant concepts of Isabelle/HOL. Next, Chapter 4 shows our proofs and definitions for parity games, while Chapter 5 shows them for tangle learning and its related concepts. In Chapter 6, we look at two related Isabelle proofs and compare them to our own. Chapter 7 discusses our results and reflects on our solution. Finally, Chapter 8 concludes the thesis and gives recommendations for future work.

Chapter 2

Background

This thesis concerns two core subjects: parity games and interactive theorem proving. We will use this chapter to discuss these topics in a broad sense to provide background for the thesis. Parity games are the main concept relevant to the tangle learning, which has the aim to solve these games. We discuss their history as well as related algorithms in Section 2.1. Interactive theorem proving is the practice of using software to prove mathematical properties. Isabelle is an example of such software. We will discuss the history of this practice, as well as Isabelle itself and the proving of algorithms using Isabelle in Section 2.2.

2.1 Parity Games

Parity games are a type of two-player game. They were first introduced by Emerson and Jutla [9] as an acceptance condition for tree automata, before they were used as a winning condition for a game. In an automaton, we have graph that consists of a set of states and properties of those states, as well as a successor function that decides successors for each state. A run is a sequence of states and an acceptance condition determines whether a run is accepted, meaning it satisfies some property. In a game, we call the acceptance condition a win condition, and a run is called a play. Instead of a fixed successor function, a game has a number of players who own a set of nodes in the graph. The objective of these players is to choose successors that cause them to win plays. Other types of games exhibit the same relation, such as Rabin games and the related Rabin automata [24].

Immediately noted in their introduction by Emerson and Jutla [9] and later reinforced by proofs like that of Björklund et al. [4] is the property of *positional determinancy*. The decisions made by a player in a parity game form a *strategy*, which restricts the moves in a play. If a player has a strategy that causes them to win all plays starting from a specific node, they have won that node. Now, positional determinancy is the fact that, if a strategy exists that wins a node, then there also exists a *positional* (or *memoryless*) strategy that wins the node. This is a strategy where every decision is independent of all prior moves in the play, meaning the chosen successor for some node is the same every time it is visited. As a result, there is also a fixed winner for every node in the graph. Furthermore, for whole regions of nodes won by a player, there exists a single strategy to win the entire region. As a consequence of this property, we can 'solve' parity games. The solution of a game consists of the regions won by each player, often accompanied by the strategies that win these regions.

We can broadly divide algorithms that solve parity games into two categories. The first of these consists of algorithms that assign a value to a node and adjust ('improve') this value, eventually linking it to which player wins the node. An example of such an algorithm is Jurdziński's small progress measures [17]. It assigns a value to each node and repeatedly increases it until it cannot change further. Succinct progress measures [18] is a quasipolynomial variant of this algorithm.

The second group of algorithms relies on the concept of attractors, which are sets of nodes from which a player can force their opponent to move to a target region. The algorithms in this category use attractors to recursively partition the game into winning regions for the players, until the regions are maximal. This recursive approach was first introduced by McNaughton [22] for games in general. It was later applied specifically to parity games and attractors by Zielonka [28]. Priority promotion [2] is another example, and tangle learning also belongs to this category.

2.2 Interactive Theorem Proving

Many mathematical proofs presented in the scientific literature are considered *informal proofs* [12, 13]. These proofs are written in natural language and do not necessarily cover the most minute details; it is up to the reader to check these proofs. Consequently, these proofs may contain unnoticed errors, and their validity is not guaranteed.

In contrast, a *formal proof* is specified in a formal language and reasons down to the fundamentals of mathematics. Clearly, a formal proof can be more rigorously validated by a reader, but the process of doing so may also be more complex as the proof itself gets more complex. Regardless, we may wish to translate an informal proof into a formal proof, a process we call *formalisation*. We may call the resulting proof a formalisation as well.

To ease the difficulty of verifying formal proofs, we frequently employ computers. In interactive theorem proving, the proof is guided by a human user who specifies a property and a (partial or entire) proof in a formal language, while the software checks the validity of the proof. The degree of automation in these checks may vary, with the software being able to prove some properties without human interference, while others require more steps to be specified by the user. This stands in contrast with automated theorem proving, where software proves properties entirely on its own. According to Harrison et al. [14], the concept of interactive theorem proving originated in the 1960's, while most work on computer-checked proofs before then focused on automated theorem proving.

There exist various interactive theorem provers, also called proof assistants, that can be used for interactive theorem proving. We will focus on Isabelle/HOL [23], which traces its lineage to the HOL family of theorem provers [14]. HOL, meaning 'higher-order logic', is the system that Isabelle/HOL bases its formal language on. It further takes its syntax from functional programming languages.

The archive of formal proofs[1] is an online repository of proofs made with Isabelle/HOL. Here, we can find proofs for some fundamental theorems such as Euclid's perfect number theorem [15]. We also find libraries, frameworks, and algorithms with accompanying correctness proofs.

We find that most algorithms for which correctness has recently been proven in Isabelle are given functional programming implementations. In some cases, an algorithm was never defined in an imperative form, such as Bergström and Weber's solver for quantified boolean formulas [3]. In other cases, an imperative algorithm has been rewritten to a functional implementation, as is the case for Rau's Earley parser [25]. This is to be expected, given Isabelle is based on functional programming paradigms.

There do exist means to formalise imperative algorithms without rewriting them as functional algorithms. The combination of imperative HOL [5] and the Isabelle Refinement Framework offer solutions for formalising imperative algorithms [19]. These are also used to generate verified implementations of algorithms.

[1] Hosted at isa-afp.org.

Chapter 3

Preliminaries

The main content of this thesis relies on specific concepts and practices. It is important for the reader to understand these. In this chapter, we give all relevant definitions and explain necessary concepts. Section 3.1 gives definitions on the subjects of parity games and tangle learning, which we will refer to in future chapters. In Section 3.2, we give a brief introduction to proofs and definitions in Isabelle, showing their basic syntax and structure.

3.1 Parity Games

We base our definitions for parity game concepts and tangle learning mainly on Van Dijk's paper [7]. Some definitions have been adjusted based information given in his Master's course *Model Checking and Parity Games* and other unpublished improvements made by Van Dijk.

A parity game is a two-player game played on a finite directed graph without dead ends. All nodes in the graph are owned by one of two players, *Even* and *Odd*. Each node has also been assigned a natural number, its *priority*. In further definitions, we will denote a relevant player as α, and their opponent as $\overline{\alpha}$. We may also write 0 for Even and 1 for Odd. Furthermore, the priority of a node v is written as $pr(v)$, and the highest priority of a set of nodes V is likewise denoted as $pr(V)$. The left-hand side of Figure 3.1 shows an example of a party game. In this and further diagrams, circular nodes belong to Even, and pentagonal nodes belong to Odd.

Definition 3.1. A *parity game* is a tuple $\mathcal{G} = (V, V_0, V_1, E, pr)$, where:

- $\mathcal{A} = (V, V_0, V_1, E)$ is an *arena*, a finite directed graph without dead ends where:
  - V is the set of all nodes in the graph
  - $V_0 \subseteq V$ is the set of nodes that belong to Even
  - $V_1 = V \setminus V_0$ is the set of nodes that belong to Odd
  - $E \subseteq V \times V$ is the set of directed edges in the graph
- $pr : V \to \mathbb{N}$ is the *priority function*, which assigns a *priority* to each node in the game

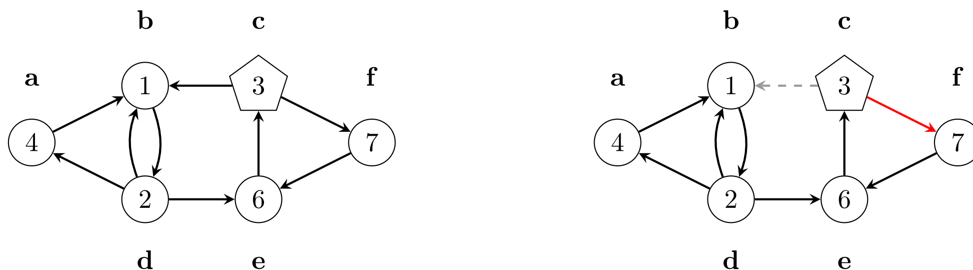

Figure 3.1: A simple parity game (left) and the induced subgame of Odd's strategy [c↦f] (right).

The game is played by moving a token along the edges of the graph. An infinite sequence of these moves is called a *play*. This play is won by Even if the highest priority that is encountered infinitely often in the play is an even number, and otherwise it is won by Odd.

Definition 3.2. A *play* $\pi \in V^\omega$ is an infinite sequence of valid moves in a parity game.

These decisions made by a player form a *strategy*. This is a partial function that assigns successors to nodes belonging to that player. We say a strategy is *complete* when it has all of a player's nodes in its domain, and that it is *valid* when it only contains legal edges from E as its moves.

Definition 3.3. A *strategy* $\sigma : V_\alpha \rightharpoonup V$ is a partial function that assigns successors in E to nodes in V_α, with those nodes belonging to α.

The successor of a node given by a strategy is written as $\sigma(v)$. If we restrict the successors in E to those given by σ, we get its *induced subgame*. Shown on the right-hand side in Figure 3.1 is the induced subgame for Odd's strategy [c↦f]; the edge (c,b) is effectively no longer part of the game.

Definition 3.4. The *induced subgame* $\mathcal{G}[\sigma]$ of σ is the parity game $\mathcal{G}$ with the successors of all v in the domain of σ restricted to $\sigma(v)$.

We call any set of nodes in a parity game (and in general graphs) a *region*. A region is *closed* if no nodes in the region have successors outside the region. Often, we say a region is closed for a player if the nodes belonging to that player have no successors outside the region. We may also say a region is *partially closed* in some region contained within it. If a region R is partially closed in $R' \subseteq R$, then all nodes in R' only have successors in R, but nodes in $R \setminus R'$ can have successors outside of R.

A region in a graph is a *strongly connected component* (SCC) if all nodes in the region can be reached by all other nodes in the region, and there exists no larger region that contains it that is also strongly connected. A *bottom* SCC, then, is a strongly connected component that is closed. We also say that a strongly connected component is *nontrivial* if it contains at least one edge; by its definition, a strongly connected component could consist of a single node without successors, which we would consider a *trivial* SCC.

3.1.1 Attractors

Attractor sets are regions within which a player can force their opponent to move to a target region. In Section 2.1 we discussed that various algorithms for solving parity games rely on this concept, including tangle learning (which uses a modified version discussed later in this section).

Definition 3.5. An attractor set $Attr_\alpha^\mathcal{G}(A)$ is the set of all nodes in V for which α has a strategy σ that will lead all plays that start in the attracted region into the target region A.

We say that A attracts $Attr_\alpha^\mathcal{G}(A)$. When we want to show that a set is an attractor, we need to produce the strategy σ that forces all plays in the attractor to its target region. We call this its *witness strategy*.

We can construct an attractor as follows: we start with the set A, and then repeatedly extend the set with all α-nodes that have at least one successor in $Attr_\alpha^\mathcal{G}(A)$, and all $\overline{\alpha}$-nodes that *only* have successors in $Attr_\alpha^\mathcal{G}(A)$, until the maximal attractor set is obtained. We may define this inductively in three steps:

- **Base:** add all nodes in the target region A.
- **Player:** add all α-nodes that have at least one successor in the attractor.
- **Opponent:** add all $\overline{\alpha}$-nodes that only have successors in the attractor.

By repeating these steps exhaustively, we obtain our attractor set. We can also use these steps to construct our witness strategy. We keep track of a strategy σ. Then, each time we perform the **player** step, we add the decision to move to some (arbitrarily chosen) successor in the current attractor to σ. Once no more steps can be taken, σ is our witness that forces all plays in $Attr_\alpha^\mathcal{G}(A)$ to move to A.

3.1.2 Dominions

Because parity games exhibit positional determinancy, there exist regions in the game that are won by one player. These are called *dominions*, or *winning regions*. We will often use the latter term, but we take their meanings to be the same.

Definition 3.6. A *dominion* D for α is a set of nodes for which player α has a strategy σ such that:

1. all plays in $\mathcal{G}[\sigma]$ stay in D, regardless of the moves of player $\overline{\alpha}$, and
2. α wins all nodes in D.

This definition is equivalent to saying that the region D is closed in $\mathcal{G}[\sigma]$, and that all cycles reachable from nodes in D won by α.

In Figure 3.2 (left), two dominions are shown: one for Even and one for Odd. These dominions are maximal for their respective players. Odd has the strategy [c↦f] and Even has two potential strategies: [d↦b], and [d↦a]. Neither player can win the entire game.

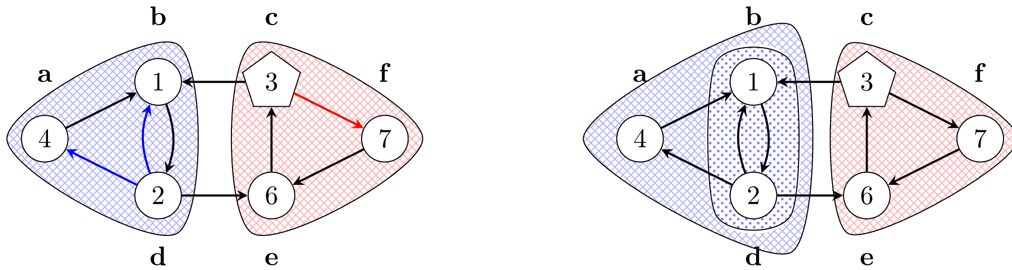

Figure 3.2: A parity game showing two maximal dominions (left), and the same game showing three tangles (right). Note the equivalence of the maximal tangles and the dominions.

3.1.3 Tangles

Based on the definition given by Van Dijk [7, Section 3], a *tangle* is the following:

Definition 3.7. A tangle for player α is a nonempty set of nodes $U \subseteq V$ if α has a strategy $\sigma : U_\alpha \to U$, such that the graph $(U, E‘)$, with $E‘ := E \cap (\sigma \cup (U_{\overline{\alpha}} \times U))$, exhibits two properties:

1. $(U, E‘)$ is strongly connected, and
2. player α wins all cycles in $(U, E‘)$.

Note that $E‘$ in this definition is effectively the same as $\mathcal{G}[\sigma]$ restricted to the region U. We will use this fact in some of our later proofs.

When we have a tangle t, we write its nodes as t and its strategy as $\sigma_T(t)$. $E_T(t) := \{v \mid u \to v \wedge u \in t \cap V_{\overline{\alpha}} \wedge v \in V \setminus t\}$ is the set of successors outside of t of $\overline{\alpha}$-nodes in t. We call these the tangle's *escapes*, as the opponent can escape from the tangle to these nodes. We also write T_0 for the set of all tangles won by Even, and T_1 for the set of all tangles won by Odd. T_α then, is the set of tangles for a relevant player α.

Van Dijk noted three observations about tangles [7, Section 3]:

1. A tangle from which the opponent $\overline{\alpha}$ cannot leave is a dominion.
2. Every dominion contains one or more tangles.
3. Tangles may contain tangles with a lower maximum priority.

The right-hand side of Figure 3.2 shows three tangles: Even has two tangles, one of which is enclosed in the other, larger tangle, and Odd has one tangle. The figure demonstrates Van Dijk's observations. Tangles on the right side are contained in dominions on the left, in this case because they are equivalent. The right side also shows that one tangle contains another with a lower maximum priority.

3.1.4 Tangle Attractors

Tangle attractors are a modified version of attractor sets. Instead of only attracting single nodes in V, it also attracts entire tangles in a set of known tangles T where $\overline{\alpha}$ can only escape into the attractor. Now, the attractor attracts nodes in V and nodes of tangles in T to A as α.

Definition 3.8. A *tangle attractor set* $TAttr_{\alpha}^{\mathcal{G},T}(A)$ is the set of all nodes in V for which α has a strategy σ that forces all plays that start in the attracted region to either move into the target region A, or be won by α.

We can once again define an inductive algorithm for creating a tangle attractor. The first steps of this algorithm are the same as those for a standard attractor, and one additional step is added:

- **Base:** add all nodes in the target region A.
- **Player:** add all α-nodes that have at least one successor in the attractor.
- **Opponent:** add all $\overline{\alpha}$-nodes that only have successors in the attractor.
- **Tangle:** add all known tangles for α from which the opponent can only escape into the attractor.

One further difference between tangle attractors and standard attractors, is that in the construction of a witness strategy, a tangle attractor will choose a successor in the attractor for every node in A that has such a successor. The tangle learning algorithm requires this step to find complete strategies for tangles.

Another important concept related to attractors is α-maximality. This concept is generally used for both standard attractors and tangle attractors, but we only need it for tangle attractors. When a region is α-maximal, it cannot be further extended by a (tangle) attractor.

Definition 3.9. A region A in a game is α-maximal when $TAttr_{\alpha}^{\mathcal{G},T}(A) = A$.

3.1.5 Tangle Learning

Van Dijk defines the *tangle learning* algorithm in two parts: the `search` algorithm (Algorithm 3.2) takes a game and a set of known tangles, searches for new tangles, and returns the newly found tangles; the `solve` algorithm (Algorithm 3.1) iteratively uses the `search` algorithm to compute the winning regions and strategies for both players. The definitions used here are different from those given in Van Dijk's paper [7], instead being based on later improvements made by Van Dijk.

Algorithm 3.1 The solve algorithm.

1: **function** SOLVE($\mathcal{G}$):
2: $W_0 \leftarrow \emptyset, W_1 \leftarrow \emptyset, \sigma_0 \leftarrow \emptyset, \sigma_1 \leftarrow \emptyset, R \leftarrow V, T \leftarrow \emptyset$
3: **while** $R \neq \emptyset$ **do**
4: $Y \leftarrow$ SEARCH(R, T)
5: $T \leftarrow T \cup \{(U,\sigma) \in Y \mid E(U_{\overline{\alpha}}) \not\subseteq U\}$
6: $D \leftarrow \{(U,\sigma) \in Y \mid E(U_{\overline{\alpha}}) \subseteq U\}$
7: **if** $D \neq \emptyset$ **then**
8: $W_0`, \sigma_0` \leftarrow TAttr_0^{\mathcal{G},T}(\bigcup D_0)$
9: $W_0 \leftarrow W_0 \cup W_0`, \sigma_0 \leftarrow \sigma_0 \cup \sigma_0`$
10: $W_1`, \sigma_1` \leftarrow TAttr_1^{\mathcal{G},T}(\bigcup D_1)$
11: $W_1 \leftarrow W_1 \cup W_1`, \sigma_1 \leftarrow \sigma_1 \cup \sigma_1`$
12: $R \leftarrow R \setminus (W_0 \cup W_1)$
13: **return** W_0, W_1, σ_0, σ_1

The `solve` algorithm (Algorithm 3.1) finds the winning regions and strategies in $\mathcal{G}$ by repeatedly searching for dominions and extending them to their maximum size using tangle attractors. The algorithm starts with empty winning regions, empty winning strategies, all nodes V in the game, and an empty set of tangles T (line 2). It then passes the current region R and T to the `search` algorithm, which searches for new tangles in R (line 4). Next, open tangles (those from which the opponent cannot escape) are added to T (line 5), and all closed dominions in the set of tangles are gathered

in D (line 6). If there are dominions, they are maximised with the tangle attractor over all of their player's dominions. The winning regions are then updated with the attracted region, and the winning strategies are updated with the attractor strategy (lines 8–12). The new winning regions are then removed from the current region before the next iteration (line 12). After the final iteration, when the remaining region is empty, the winning regions and strategies are returned (line 13).

Algorithm 3.2 The search algorithm.

1: **function** SEARCH(R, T)
2: $\quad Y \leftarrow \emptyset$
3: $\quad$ **while** $R \neq \emptyset$ **do**
4: $\quad\quad p \leftarrow pr(R), \alpha \leftarrow pr(R) \mod 2$
5: $\quad\quad A \leftarrow \{v \in R \mid pr(v) = p\}$
6: $\quad\quad Z, \sigma \leftarrow TAttr_{\alpha}^{R,T}(A)$
7: $\quad\quad O \leftarrow \{v \in A_{\alpha} \mid E(v) \cap Z = \emptyset\} \cup \{v \in A_{\overline{\alpha}} \mid E(v) \cap R \not\subseteq Z\}$
8: $\quad\quad$ **if** $O = \emptyset$ **then** $Y \leftarrow Y \cup sccs(Z, \sigma)$
9: $\quad\quad R \leftarrow R \setminus Z$
10: $\quad$ **return** Y

The `search` algorithm (Algorithm 3.2) finds new tangles. The algorithm tracks newly discovered tangles in the set Y (line 2), and starts by setting Y as empty. The algorithm then enters a loop, which continues until the current region R is empty (line 3). At the start of each iteration it obtains the highest priority p in the region and the player α who wins this priority (line 4). Next, it tangle attracts to the set of all nodes in R with priority p, storing the resulting region in Z and the strategy in σ (lines 5–6). After this, it gets all 'open' nodes that were attracted to. These are the nodes in region A that can escape the attracted region Z, either by being α-nodes that cannot stay in Z, or by being $\overline{\alpha}$-nodes that can leave Z (line 7). If no open nodes exist, then the tangles in Y are updated with the bottom SCCs of Z restricted by the strategy σ (line 8). The method for obtaining the bottom SCCs can be any appropriate algorithm, such as Tarjan's algorithm [27]. Next, the region Z is removed from the current region R before the next iteration (line 9). After the last iteration, the updated set of tangles Y is returned (line 10).

In Van Dijk's original definition, the obtaining of new tangles in Z consisted of an extra step, which was the removal of the open nodes [7, Section 4.3]. The new version of the algorithm omits this step to simplify the algorithm, which is a variant suggested in Van Dijk's paper [7, Section 4.6]. After performing that step, the algorithm would continue with finding the bottom SCCs as in line 8 of the updated version. Another major difference is that the `search` algorithm no longer returns a tangle that is a dominion, which would have been used by the `solve` algorithm instead of the dominions it now finds itself in line 6 of Algorithm 3.1.

3.2 Definitions and Proofs in Isabelle

The proofs and definitions written for this thesis were written with Isabelle/HOL. Isabelle uses a Haskell-like functional programming syntax combined with mathematical notation. A basic understanding of this syntax is essential for reading the results displayed in the thesis. Therefore, we will now cover relevant elements of the syntax of Isabelle/HOL. These explanations are based on the documentation for Isabelle/HOL[1].

Isabelle proofs may become long and difficult to read. When we present our results, we aim to show the most important definitions, as well as proofs that can be presented in a readable fashion. For longer proofs, we will instead explain our steps as a traditional pen-and-paper proof.

3.2.1 Theories

Isabelle proofs are contained in so-called 'theories', which are files that are often given the `.thy` extension. The `theory` keyword starts a theory, which must have a name matching its filename. This is followed by an `imports` that lets it import other theories to use their lemmas and definitions in its own proofs. The theory then begins with the `begin` keyword, and ends with the `end` keyword. Every

[1] Available at isabelle.in.tum.de/documentation.html.

theory must import at least one other theory, frequently `Main`, a theory that contains the standard library of Isabelle/HOL, or some other theory that imports `Main`. The outline of a theory looks as follows:

```
theory SquareRoots
imports Main
begin
...
end
```

Listing 3.1: The general structure of an Isabelle theory.

3.2.2 Datatypes

Isabelle allows us to define datatypes that we can operate on. For these datatypes, we define a number of cases that represent possible values of that type. A simple example would be a Boolean which can be true or false. We can define this datatype as follows:

```
datatype boolean = TRUE | FALSE
```

Listing 3.2: A Boolean datatype defined in Isabelle/HOL.

Datatypes can also be recursive, allowing for more complex data to be represented. This ranges from simple recursion, where only one case is recursive, to more complex recursions that allow for the definition of types such as trees. Lists are a classic example of a recursive datastructure in functional programming. Typically, they contain an empty case, Nil, and a case containing a value and a consecutive list:

```
datatype 'a list = Nil | Cons 'a "'a list"
```

Listing 3.3: A list defined in Isabelle/HOL.

Note that this definition references a generic datatype `'a`. This could be any other datatype, such as a natural number: a list of natural numbers would be a `nat list`.

Finally, it is possible to rename an existing datatype or combine existing datatypes using the `type_synonym` keyword. This allows us to use relevant names in the context of our proofs. For instance, we may want to represent a directed graph as a set of edges, combining a pair and a set:

```
type_synonym 'a dgraph = ('a × 'a) set
```

Listing 3.4: A directed graph as a set of directed edges in Isabelle/HOL.

3.2.3 Lists

We frequently use lists in our definitions. The standard library definition for lists is similar to Listing 3.3. However, it comes with various syntactic shortcuts that we have not yet covered. First of all, we may write an empty list as simply `[]`, equivalent to `Nil`. Furthermore, a list with some number of elements can be written as `x#xs`, where `x` is the 'head' of the list: the first element in the list. `xs` is the 'tail' of the list: the remainder of the list after `x`. A single-element list might now be written as `x#[]`, equivalent to `Cons x Nil`. Finally, we may also write a list as $[x_1,x_2,x_3]$ to shorten $x_1\#x_2\#x_3\#[]$.

There are a few common operations on lists that we will encounter in our formalisations. The first is the appending of one list to another: to append `ys` to the end of `xs`, we write `xs@ys`. Another common operation is getting the length of a list, simply written as `length xs`. Finally, we can obtain the set of elements in a list using `set xs`. We will cover sets next.

3.2.4 Sets

Sets are commonly used in our definitions. They represent the mathematical concept of a set: they are unordered collections of a potentially infinite number of unique elements. The most basic set we will often use is the empty set $\emptyset$. In Isabelle, this is written as `{}`. Besides this, sets come with a number of predefined operations such as membership (`x ∈ S`), insertion (`insert x S`), union (`A ∪ B`),

intersection (`A ∩ B`), and subtraction (`A - B`). We see that the syntax for these is similar to standard mathematical notation, with some exceptions.

Another common operation is defining a set of elements that meet certain conditions. For instance, `{x. even x}` gives us the set of all even numbers. `{x | x y. (x,y) ∈ E}` is a more complicated syntax. It gives us all nodes in a graph E that have a successor, specifically those for which there exists a y such that the edge (x, y) exists in E. Variables specified after the bar '`|`' are treated as existential (∃) quantifiers.

3.2.5 Nonrecursive Definitions

We can define operations on datatypes that do not contain recursion using the `definition` keyword. These usually check for some mathematical property. For example, we could define an operation `nonemptysubset` that confirms whether a set A is a nonempty subset of B:

```
definition nonemptysubset :: "'a set ⇒ 'a set ⇒ bool" where
  "nonemptysubset A B ≡ A ≠ {} ∧ A ⊆ B"
```

Listing 3.5: A nonrecursive definition for checking if A is a nonempty subset of B.

In the Isabelle syntax, the name of the definition immediately follows the `definition` keyword. After this, we have `::`, followed by the type of the definition. In this case, it takes two sets as input, and its result is a `bool`. After the type definition, we have the `where` keyword, which is followed by the actual definition. This definition states that `nonemptysubset` for A and B is equivalent to a check whether A is not the empty set, and is a subset of B.

3.2.6 Recursive Functions

Simple recursive functions can be defined with `fun`. These functions follow common functional programming conventions, where we may define a number of cases for the function to match. A simple example would be to sum all elements in a list of natural numbers:

```
fun addlist :: "nat list ⇒ nat" where
  "addlist [] = 0"
| "addlist (x#xs) = x + addlist xs"
```

Listing 3.6: A recursive definition for summing a list of natural numbers in Isabelle/HOL.

Similarly to nonrecursive definitions, we follow the `fun` keyword with the name of the function and the type. After the `where` keyword, we define our cases for the function. If we add an empty list, we get 0. This case is required because it is the point where our recursion ends. Our second case has a list with head x and tail xs. This case adds the value x to the result of a recursive call to `addlist`.

Defining cases like this is a common element of functional programming languages, known as *pattern matching*. Each case shows how the function responds to a certain input pattern, and we must define all necessary cases to show that the recursion eventually terminates.

The `fun` construct automatically tries to prove termination for us. Furthermore, Isabelle automatically generates rules and proofs for some properties of the function, which will be useful in proofs related to the definition. One example is the induction rule it generates for the function. This rule can be used for a *structural induction*: an induction over the patterns of the function. In this case, it would be an induction with a case for the empty list, and a case for a list with a head and a tail, which are the same cases as those usually used for an induction on a list.

We must note that `fun` is not exclusively used for recursive functions. We may also use its pattern matching to simply define different cases for different inputs. We may, for instance, use it to invert a Boolean[2]:

```
fun invert_bool :: "bool ⇒ bool" where
  "invert_bool True = False"
| "invert_bool False = True"
```

Listing 3.7: A nonrecursive function defined using `fun`.

[2]There are much simpler ways to invert Booleans. This serves merely as a simple example.

3.2.7 Inductive Definitions

Functions may also be defined inductively, which is often done for inductive predicates. These are also defined using a series of cases. We may see the difference between these and recursive definitions as follows: a recursive definition 'breaks down' a structure, for example by working through a list until it is empty; on the other hand, an inductive definition 'builds up', adding on to itself.

`inductive` definitions have cases that usually consist of preconditions for a predicate to be true. For example, we can define reachability in a directed graph in two cases:

```
inductive reachable :: "'a ⇒ 'a ⇒ bool" where
  refl: "reachable x x"
| trans: "(x,y) ∈ E ⟹ reachable y z ⟹ reachable x z"
```

Listing 3.8: An inductive definition of reachability in a directed graph.

Here, the `refl` case expresses the reflexivity of reachability: a node is trivially reachable from itself. The `trans` case expresses transitivity: if we have an edge from x to y, and z is reachable from y, then z is also reachable from x.

Like with recursive functions, Isabelle generates rules and proofs related to the definition. For instance, an induction rule is generated that follows the cases of the definition.

A related construct is the `inductive_set`. This allows us to inductively define set construction. For example, we might get the set of all reachable nodes from some node t:

```
inductive_set reachable_nodes :: "'a ⇒ 'a set" for t where
  base: "t ∈ reachable_nodes t"
| step: "(x,y) ∈ E ⟹ x ∈ reachable_nodes t ⟹ y ∈ reachable_nodes t"
```

Listing 3.9: An inductive set definition for the set of all nodes reachable from some t.

The `base` case says that t is reachable from itself. The `step` case says that we can add y to the reachable nodes from t if there is an edge from some x that is reachable from t to y.

Internally, the `inductive_set` uses an `inductive` definition. We could obtain the same result if we exhaustively apply an inductive predicate, but we benefit here from the definition doing it automatically. The inductive set exhaustively performs steps, yielding a maximal set.

3.2.8 Contexts and Locales

We can use a `context` to set overarching assumptions and variables for any proofs and definitions contained within it. We might fix a graph E in some context, and have our definitions operate on E without explicitly requiring it as input. Furthermore, we could set assumptions about E, such as it being finite. This assumption can then be used in any proof in the context.

A definition that exists in a context with fixed variables can be used outside that context by supplying that variable to it. Similarly, if a proven property inside a context depends on an assumption of that context, we can only use it outside of that context if we also show that assumption to be true.

A `locale` is much like a context. It can have fixed assumptions and variables. Additionally, we can give a locale a name. Using that name, we can then open a context with the same assumptions by simply referring to that name. This allows us to come back to locales later and show additional properties. Furthermore, a locale can be extended by other locales, letting us slowly build definitions in progressively more specific locales.

3.2.9 Lemmas, Theorems, and Corollaries

Any property we wish to prove in Isabelle can be expressed as a `lemma`, `theorem`, or `corollary`. These three names are interchangeable, having no effect on how the property is interpreted by the software. A property starts with one of these keywords, followed by an optional name. Next, we can directly state our property:

```
lemma empty_list_length: "length [] = 0"
```

Listing 3.10: A basic property expressed in Isabelle/HOL.

We may also encode assumptions in our property, where the assumption leads to our conclusion. An assumption is placed before the conclusion, with $\Longrightarrow$ in between:

```
lemma nonempty_list_length: "xs ≠ [] ⟹ length xs > 0"
```

Listing 3.11: A property with an assumption.

Multiple assumptions can be chained, placing $\Longrightarrow$ between each assumption. However, a shorter syntax for multiple assumptions is available, where we group assumptions. Double brackets outline the group, and semicolons separate the assumptions:

```
lemma nonempty_lists_append_length: "⟦xs ≠ []; ys ≠ []⟧ ⟹ length (xs@ys) ≥ 2"
```

Listing 3.12: Grouping assumptions with double brackets.

Another way to define properties is indirect. After fixing the name, we can set assumptions with the assumes keyword. These assumptions can optionally have a name. Our conclusion is given after the shows keyword:

```
lemma nonempty_lists_append_length:
  assumes xs_nonempty: "xs ≠ []"
  assumes ys_nonempty: "ys ≠ []"
  shows "length (xs@ys) ≥ 2"
```

Listing 3.13: The alternative syntax for defining properties.

The proof of a property immediately follows its definition. There are two ways to prove properties: automated provers and Isar proofs. We will cover these methods of proving in the next sections.

3.2.10 Automated Proofs

We can prove relatively simple properties using the automated provers built into Isabelle. The most common of these is `auto`, which applies certain lemmas and rewriting rules from the standard library and is able to perform basic set theory and logical reasoning to prove goals. A less aggressive version of `auto` is `simp`. Some other provers include `blast`, `fast`, `force`, and `fastforce`, which each have their own specialisations for proving properties. In practice, the choice of automated prover for a property is often arbitrary or based on which prover can prove a property in the shortest amount of time.

For our example, we return to our inductive set `reachable_nodes` (Listing 3.9). We want to prove that this set is equal to the reflexive transitive closure E^* of E applied to t. Instead of defining this all at once, we will prove each direction independently. First, we use automated provers to prove that, if x is in the set of nodes reachable from t, then the edge (x, t) is part of the reflexive transitive closure E^*:

```
lemma reachable_nodes_impl_rtrancl: "x ∈ reachable_nodes t ⟹ (t,x) ∈ E*"
  apply (induction rule: reachable_nodes.induct)
  apply (auto)
  done
```

Listing 3.14: Our first lemma for reachable_nodes with its automated proof.

Our proof uses an induction based on the definition of `reachable_nodes`, which was generated automatically. We invoke this with a `rule` parameter. Next, it completes the proof using `auto`. Invoking any rule or prover is done using the apply keyword, and we terminate the proof with the done keyword.

Next, we prove the other direction. We say that if an edge (t, x) is in the reflexive transitive closure E^* of E, then x is in the set of reachable nodes:

```
lemma rtrancl_impl_reachable_nodes: "(t,x) ∈ E* ⟹ x ∈ reachable_nodes t"
  apply (induction rule: rtrancl_induct)
  subgoal by (simp add: reachable_nodes.base)
  subgoal by (simp add: reachable_nodes.step)
  done
```

Listing 3.15: Our second lemma for reachable_nodes with its automated proof.

Our proof uses the induction rule `rtrancl_induct` from the standard library. This rule gives us two statements to prove: that t is in its own reachable nodes, and that if we have a reachable node y with a successor z, z is also reachable. We can focus on a single statement using the subgoal

keyword. The first subgoal is solved by adding the automatically generated rule for the `base` case of `reachable_nodes` to the known rules of `simp`. The second is solved by adding the rule for the `step` case. Normally, we would have to end a proof of a `subgoal` with `done`, but the `by` keyword lets us finish it in a single step, without needing `done`.

We can now use the former lemmas as auxiliary lemmas for our final proof with the `using` keyword. This allows us to prove the final lemma in a single step. We use the `‘‘` operator to apply the closure to the set $\{t\}$, which gives us all reachable nodes from t, and show that this is equal to the `reachable_nodes` of t:

```
lemma reachable_nodes_is_rtrancl: "reachable_nodes t = E* ‘‘ {t}"
  using reachable_nodes_impl_rtrancl rtrancl_impl_reachable_nodes by blast
```

Listing 3.16: Our automated proof for equivalence of reachable_nodes and E^*.

One more keyword of note is the `unfolding` keyword. It is placed before a step, similar to the `using` keyword. Generally, we use it to replace part of a proposition that was given in a `definition` by its actual meaning, 'unfolding' it. This can be done by unfolding its automatically generated `[name]_def` rule. However, it can be used in practice to replace instances of the left-hand side of a known equality with its right-hand side. Alternatively, rewriting equalities this way can be achieved by adding the equality to the rules used by `simp`, though this may have other side-effects.

Automated provers can show relatively simple properties quickly. However, as properties get more complex, so do the automated proofs. They might also be unable to prove certain properties. We will look at a different approach for complex proofs next.

3.2.11 Isar Proofs

For complex proofs, Isabelle has the Isar language. The syntax of Isar emulates step-by-step proofs on paper, with propositions following from previous propositions, eventually showing a thesis. This breaks down complex proofs into smaller steps that can be understood by a reader and proven by the automated provers.

An Isar proof starts with the `proof` keyword. This keyword may be followed by some rule to apply initially. We may for instance do a proof by contradiction using `proof (rule ccontr)`. If no rule is specified, Isabelle automatically applies some obvious rule for the property it aims to prove, but inserting a `-` after `proof` prevents it from doing so.

If a proof depends on assumptions, they must be specified at the start of the proof with the `assume` keyword. The exception are assumptions specified with the `assumes` keyword, as these are automatically taken as assumed by the proof. If any assumptions are missing, a proof cannot be completed.

A proposition inside a proof is stated with a `have` keyword. To name a proposition, we simply enter its name and a colon before the statement: `have x_eq_2: "x = 2"`. If a proposition follows from some previous (named) propositions or assumptions, we have several options for referencing them. First of all, these can be referenced first with the `from` keyword: `from x y have z`. Names may also be omitted by restating the literal fact being referenced: `from <x=2> have "even x"`. Furthermore, if a proposition follows directly from the previous proposition, we may use the `hence` keyword, skipping `from` and `have`, but this would not allow us to reference additional statements. To reference multiple statements in addition to the prior one, we can use the `with` keyword, and use `have` as normal after the list of statements. Finally, we could use a series of propositions to prove a later one by placing `moreover` before each subsequent proposition we prove, and `ultimately` before the proposition that will use them.

At the end of a proof, we want to show one or multiple properties to be true to prove our thesis. The `show` keyword allows us to mark a proposition as a goal for the proof. There is also an equivalent of `hence` for these goals, `thus`, which makes the goal follow from the previous proposition. The actual proposition can either be typed out entirely, but in many cases where the thesis is clear, we can take a shortcut by typing `?thesis`. After we have shown all proof goals, we end the proof with `qed`.

To put this into perspective, we show an example of a simple proof. Say we want to show that when our `addlist` function (Listing 3.6) returns a number greater than zero, the list must be longer than zero elements. We can do this with a single intermediate step:

```
lemma "addlist xs > 0 ⟹ length xs > 0"
proof -
  assume "addlist xs > 0"
  hence "xs ≠ []" by auto
  thus ?thesis by blast
qed
```

Listing 3.17: A simple Isar proof for addlist.

There are various other keywords and constructs in Isar proofs that we will encounter. First of all, `consider` lets us define a case distinction that we can use to prove a subsequent proposition. Furthermore, we can use `obtain` to obtain a witness from a proven existential, and we can `fix` variables and `assume` their properties to prove universal quantifiers. We can also specify definitions within proofs: `define` lets us give definitions within a proof, which we must unfold much like other definitions; while `let` also allows us to give definitions, except these are treated as shorthand variables that do not need to be unfolded. We may also structure a case distinction or an induction proof with named cases. We will not go into further detail about these constructs here; their use will be explained when we encounter them.

Chapter 4

Formalisation of Parity Games

Without parity games, there can be no tangle learning. Therefore, we must define and prove properties of parity games themselves before we can begin formalising tangle learning. In this chapter, we first discuss the basic concepts we have defined in Section 4.1. This is a brief overview, as these definitions are not the primary subject of our research. Next, we highlight some specific areas that proved difficult or have particular importance in our formalisation. Section 4.2 looks at how we can represent plays in our formalisation. In Section 4.3, we discuss attractors and difficulties encountered in their formalisation. Finally, we look at a proof for the positional determinancy of our definitions of parity games in Section 4.4.

4.1 Basic Concepts

To represent parity games, we use a series of locales that build on one another to give us progressively more specific contexts for our definitions and proofs. This has the benefit of letting us prove properties given only the relevant circumstances without imposing additional restrictions.

4.1.1 Directed Graphs

The basic element in our locales is a directed graph. The definition for directed graphs and paths within them are found in the Isabelle file `Digraphs.thy`. The graph itself does not exist in a locale, as it is a basic data type. Our definition uses a set of directed edges, which we represent as a set of node pairs. Isabelle/HOL has an existing name for such a structure: a `rel`, or relation. We use this construct for our formalisation:

```
type_synonym 'v dgraph = 'v rel
```

Listing 4.1: Our directed graph as a set of directed edges.

We also define strong connectivity for a graph (V, E). Along with strongly connected components (discussed in the next subsection), it can be found in `StrongConnectivity.thy`. We say that a graph is strongly connected if it is nonempty; E only contains nodes from V; and if, for all pairs of nodes v and v' in V, v' is reachable from v in E. For this reachability, we use the reflexive transitive closure E^* of E. This gives us our definition:

```
definition strongly_connected :: "'v dgraph ⇒ 'v set ⇒ bool" where
  "strongly_connected E V ≡ V ≠ {} ∧ E ⊆ V×V ∧ (∀v ∈ V. ∀v' ∈ V. (v,v') ∈ E*)"
```

Listing 4.2: Our definition for a strongly connected graph.

We fix a graph in a context to define paths, cycles, and lassos (Listing 4.3). Our paths consist of a start node, a list of intermediate nodes along the path, and a destination node. Next, cycles are nonempty paths from a node to itself. Finally lassos consist of a (potentially empty) path from some node x to another node, from which there is a cycle. We prove useful utility lemmas for these concepts in the same context, such as an equivalence between our paths and the reflexive transitive closure E^*, which we use to show that there are paths between all nodes in a strongly connected graph.

```
context
  fixes E :: "'v dgraph"
begin

fun path :: "'v ⇒ 'v list ⇒ 'v ⇒ bool" where
  "path v [] v' ⟷ v = v'"
| "path v (x#xs) v' ⟷ (∃y. x=v ∧ (v,y) ∈ E ∧ path y xs v')"

definition cycle :: "'v ⇒ 'v list ⇒ bool" where
  "cycle v xs ≡ path v xs v ∧ xs ≠ []"

definition reachable_cycle :: "'v ⇒ 'v list ⇒ bool" where
  "reachable_cycle x ys ≡ ∃xs y. path x xs y ∧ cycle y ys"

definition lasso :: "'v ⇒ 'v list ⇒ 'v list ⇒ bool" where
  "lasso x xs ys ≡ ∃y. path x xs y ∧ cycle y ys"

definition lasso' :: "'v ⇒ 'v list ⇒ bool" where
  "lasso' x vs ≡ ∃xs ys. vs=xs@ys ∧ lasso x xs ys"

end
```

Listing 4.3: Isabelle definitions of paths, cycles, and lassos.

4.1.2 Fixing a Finite Number of Nodes

Next, we want to fix the nodes of a graph, so that it is one with edges E and nodes V, and the graph is finite. This is our first locale, `finite_graph_V` (Listing 4.4). It fixes an E and a V, and assumes that E only contains nodes in V, and that V is finite.

```
locale finite_graph_V =
  fixes E :: "'v dgraph"
  fixes V :: "'v set"
  assumes E_in_V: "E ⊆ V × V"
  assumes fin_V[simp,intro!]: "finite V"
```

Listing 4.4: Our locale for a finite graph with edges E and nodes V.

In this locale, we define some further utility lemmas for paths, cycles, and lassos, such that they always stay in V. We also define strongly connected components (SCCs) in this locale. An SCC is a region R in V, such that the graph $(R, E \cap (R \times R))$ is strongly connected, and there does not exist a bigger region R' that contains R which is also strongly connected. We extend SCCs with bottom SCCs, which are SCCs that do not have edges leaving them (they are closed), and finally nontrivial bottom SCCs, which are bottom SCCs that contain at least one edge. These are each accompanied by utility lemmas, including a proof that every finite graph contains at least one bottom SCC.

4.1.3 Graphs Without Dead Ends

The locale `finite_graph_V_succ` (Listing 4.5) extends `finite_graph_V` with the extra assumption that every node must have at least one successor: there are no dead ends. We prove a few additional lemmas in this context; namely, we show that this lets us obtain a path of any length, as we can find a successor for any node. Consequently, we can always get a cycle in this graph, as well as a lasso.

```
locale finite_graph_V_succ = finite_graph_V E V
  for E :: "'v dgraph" and V :: "'v set" +
  assumes succ: "v ∈ V ⟹ E `` {v} ≠ {}"
```

Listing 4.5: Our locale for a finite graph without dead ends.

4.1.4 Strategies and Arenas

Strategies and arenas are defined in `Strategies.thy`. We define a strategy as a partial function that takes any node of type `'v` as input, and optionally outputs a successor for that node (Listing 4.6). Isabelle/HOL has an equivalent type for this, the `map`, which is itself a function that takes an input and outputs an `option`. `option` is a construct that lets us return `Some 'v` or `None`, with the latter representing no output.

```
type_synonym 'v strat = ('v,'v) map
```

Listing 4.6: Our data type for strategies.

We also give a definition for getting all edges included in the strategy, `E_of_strat`. This is expressed as the set of all pairs (u, v) where the strategy yields `Some v` for input u.

```
locale arena = finite_graph_V_succ +
  fixes V₀ :: "'v set"
  assumes V₀_in_V: "V₀ ⊆ V"
```

Listing 4.7: Our locale for an arena.

We introduce the sets of nodes belonging to each player in the locale `arena` (Listing 4.7), which extends the previous locale `finite_graph_V_succ`. It adds the set $V_0 \subseteq V$ for the Even player. Inside this locale, we define $V_1 = V \setminus V_0$ for the Odd player. We then introduce definitions related to strategies: `strategy_of` checks whether a strategy is valid for some player by checking if its edges are all in the graph, and its domain all belongs to a player's nodes; while `induced_subgraph` gives us the graph induced by a strategy, being all edges in E that obey the strategy:

```
definition induced_subgraph :: "'v strat ⇒ 'v dgraph" where
  "induced_subgraph σ ≡ E ∩ (((-dom σ) × UNIV) ∪ E_of_strat σ)"
```

Listing 4.8: A strategy's induced subgraph.

This definition takes a σ and combines all edges from outside of its domain to anywhere in the universe of the type `'v` with the edges of σ. It takes the intersection with E to ensure all edges in the new graph exist in E.

4.1.5 Parity Games and Players

With all previous locales defined, we can finally define parity games themselves with the locale `paritygame` (Listing 4.9), located in the file `ParityGames.thy`. We extend `arena` and add our priority function $pr : V \to \mathbb{N}$. Inside the locale we define `pr_list` to get the highest priority for a list of nodes, and `pr_set` to get the highest priority for a set of nodes. This is all we define for parity games initially.

```
locale paritygame = arena E V V₀
  for E :: "'v dgraph" and V V₀ :: "'v set" +
  fixes pr :: "'v ⇒ nat"
...

locale player_paritygame = paritygame E V V₀ pr
  for E :: "'v dgraph" and V V₀ :: "'v set" and pr :: "'v ⇒ nat" +
  fixes Vα :: "'v set"
  fixes winningP :: "nat ⇒ bool"
  assumes Vα_subset: "Vα ⊆ V"
```

Listing 4.9: Locales for a parity game and a parity game with a fixed player.

For some proofs, it is easier to work with a fixed, arbitrary player. We therefore define an extension of `paritygame`, `player_paritygame` (Listing 4.9), which fixes a player's nodes V_α, and their winning function `winningP`. Many of our future definitions and proofs are first made in this locale, after which they are introduced in `paritygame`.

To take proofs and definitions from `player_paritygame` and reintroduce them in `paritygame`, we first need to define our datatype for a player. This is done outside a locale. There are two possibilities, Even and Odd:

```
datatype player = EVEN | ODD
```

Listing 4.10: Our data type for players.

We define the utility function `opponent` to give us a player's opponent and prove that applying it twice gives us our original player. We also define `player_wins_pr` to give us the player that wins a priority. Finally, we define the function `player_winningP` to give us the winning function for each player, namely the Isabelle/HOL function `even` for `EVEN`, and `odd` for `ODD`.

Now we can link our `player_paritygame` to `paritygame`. Within the `paritygame`, we introduce two *sublocales*, which allow us to wrap locales inside an existing locale. We need to assign an E, V, V_0, pr, V_α, and `winningP` to these sublocales and then prove that they are valid instances of `player_paritygame`. The first four are the same for each, while the locale for Even gets V_0 as its V_α and `even` as its `winningP`, and the locale for Odd gets V_1 and `odd`:

```
context paritygame begin

sublocale P0: player_paritygame E V V₀ pr V₀ even
  apply unfold_locales
  by (auto simp: V₀_in_V)

sublocale P1: player_paritygame E V V₀ pr V₁ odd
  apply unfold_locales
  by auto

end
```

Listing 4.11: Introducing sublocales for each player in a parity game.

If we now have a definition in `player_paritygame`, we can add it to `paritygame` using a function. We can then use lemmas from our sublocales to prove lemmas in `paritygame`. This is a pattern that is used for many of our definitions and proofs. We will discuss winning regions in detail in 4.1.6, but we shall use them as an example here. Our definition for winning regions in `player_paritygame` is called `player_winning_region`. We can use it to define `winning_region` as follows:

```
fun winning_region where
  "winning_region EVEN = P0.player_winning_region"
| "winning_region ODD = P1.player_winning_region"
```

Listing 4.12: Using a function to bring winning regions for players into the general parity game.

One of the properties of a winning region is that it is a subset of V. We have a lemma proving this for our `player_winning_region`. If we now want to prove the same for `winning_region`, we use our lemmas for the player version and a case distinction on which player the region belongs to:

```
lemma winning_region_in_V: "winning_region α W ⟹ W ⊆ V"
  using P0.player_winning_region_in_V P1.player_winning_region_in_V
  by (cases α) auto
```

Listing 4.13: Using lemmas from a sublocale to prove properties in the main locale.

Finally, we define `V_player` within `paritygame`, as well as `V_opponent`, which give us the nodes of a player and a player's opponent, respectively. With these, we can express properties that relate to a specific player's nodes outside of `player_paritygame`.

4.1.6 Winning Regions

Winning regions are the regions within which a player has a winning strategy. We define them in the file `WinningRegions.thy`. They are placed in the `player_paritygame` locale. Our definition (Listing 4.14) of a winning region is the following: a winning region is a region W in V that is closed for $\overline{\alpha}$, and where α has a complete strategy σ in W, and has all reachable cycles from W in the subgraph won by α.

```
definition player_winning_region :: "'v set ⇒ bool" where
  "player_winning_region W ≡ W ⊆ V ∧ E `` (W ∩ V_ᾱ) ⊆ W ∧
    (∃σ. strategy_of V_α σ ∧ dom σ = V_α ∩ W ∧ ran σ ⊆ W ∧
      (∀v∈W. ∀xs. reachable_cycle (induced_subgraph σ) v xs
        ⟶ winning_player xs)"
```

Listing 4.14: Our definition of a winning region (dominion).

We also define the closely related concept of a node being won (Listing 4.15). To do this, we say that a node is won by the player α if it is in V, and there exists a strategy σ for α such that all reachable cycles from that node in the induced subgraph of σ are won by α. We then prove that each node in a winning region is won by the player.

```
definition won_by_player :: "'v ⇒ bool" where
  "won_by_player v ≡ v ∈ V ∧ (∃σ. strategy_of V_α ∧
   (∀xs. reachable_cycle (induced_subgraph σ) v xs ⟶ winning_player xs))"
```

Listing 4.15: Our definition for a node being won by a player.

We define the dual of `winning_player` as `won_by_opponent`, as well as the dual of `winning_region` as `losing_region`. We use these within the locale to ensure that only one player can win each node. Now we can prove that if a node is won by a player, it cannot also be won by their opponent. We will this lemma in Section 4.4.

Lemma 4.1. *When a node is won by one player, it cannot also be won by the other player.*

Proof. We will follow our Isabelle proof for this lemma. This proof starts by unfolding definitions, and then invoking `clarsimp` to simplify our proof obligation We must now prove that, if we have σ for α, and σ' for $\overline{\alpha}$, with all reachable cycles from v in $\mathcal{G}[\sigma]$ being won by α, then there must always exist one cycle reachable from v in $\mathcal{G}[\sigma']$ that is won by α. With this obligation, we start our Isar proof by defining the two aforementioned subgames, before giving our assumptions.

```
lemma winning_v_exclusive: "won_by_player v ⟹ ¬won_by_opponent v"
  unfolding won_by_player_def won_by_opponent_def
proof clarsimp
  fix σ σ'
  define Gσ where "Gσ = induced_subgraph σ"
  define Gσ' where "Gσ' = induced_subgraph σ'"
  assume σ_player: "strategy_of V_α σ"
    and σ_win: "∀xs. reachable_cycle Gσ v xs ⟶ winning_player xs"
    and σ'_opp: "strategy_of V_ᾱ σ'"
    and "v∈V"
```

We can use the `interpret` keyword to show that a combination of parameters satisfies the conditions of a locale. Doing so lets us use the lemmas within this locale. We do this here for the intersection of $\mathcal{G}[\sigma]$ and $\mathcal{G}[\sigma']$. In an auxiliary lemma we use here, we have proven that the intersection of two induced subgraphs belonging to opposing players is a valid parity game.

```
  interpret Ginter: paritygame "Gσ∩Gσ'" V V_0 pr
    using ind_subgraph_inter_opposed[OF Gσ_def Gσ'_def σ_player σ'_opp] .
```

As previously stated, a parity game always contains at least one cycle. We use this to obtain a cycle in the intersected graph. If a cycle exists in the intersection of two graphs it exists in both graphs, so the cycle exists in $\mathcal{G}[\sigma]$, which means it is won by the player by our earlier assumption. The cycle also exists in $\mathcal{G}[\sigma']$, meaning not all cycles reachable from v in $\mathcal{G}[\sigma']$ are won by $\overline{\alpha}$.

```
  from Ginter.cycle_always_exists[OF <v∈V>]
  obtain xs where xs: "reachable_cycle (Gσ∩Gσ') v xs" by blast
  from xs have "reachable_cycle Gσ v xs"
    using reachable_cycle_inter by fastforce
  with σ_win have "winning_player xs" by blast
  moreover from xs have "reachable_cycle Gσ' v xs"
    using reachable_cycle_inter by fastforce
  ultimately show "∃xs. reachable_cycle Gσ' v xs ∧ winning_player xs" by blast
qed
```

This concludes our proof of the fact that, if a node is won by one player, it cannot also be won by the other player. □

4.2 Plays

Recall from Definition 3.2 that a play is an infinite sequence of moves. In practice, this is a path of infinite length on the game's directed graph. Generally speaking, infinities may be hard to work with, and may not even be possible to work with in practical applications. Concrete implementations of algorithms are limited by the available memory and computing power, which makes implementing infinities difficult or impossible. Although Isabelle allows us to work on a more theoretical level, infinities still pose difficulties.

We identified three approaches to defining our infinite paths in Isabelle. The first of these is the use of a `codatatype`. This is a construct in Isabelle that allows us to define potentially infinite datastructures. Functions on codatatypes must be *corecursive*, and inductions on these types must be *coinductive*. Working with these constructs proved difficult, so this approach was not taken

Another approach is using ω-words, represented as functions over natural numbers to some type representing the alphabet of an automaton. In this case, the alphabet is the type of the nodes in the game's graph. There exists a definition of these words in the Isabelle/HOL standard library. However, this approach was also not taken.

The final approach, and the one we used, was to represent plays as lassos. As illustrated in Figure 4.1, a lasso is a path leading into a cycle. We say lassos have a 'spoke', the initial path; and a 'loop', the cycle on the end of that path. The spoke is an optional part of the lasso: if we have only a cycle, this is still a valid lasso.

We can use finite lassos to avoid working with infinite lists thanks to the properties of parity games. Because parity games are played on finite graphs, any play must eventually return to a node that has already been visited. Our strategies are positional, so once a previously visited node is reached, every subsequent move will be the same as it was when a node was visited before. Therefore, any play will take the form of a lasso with a finite spoke and an infinitely repeated loop. By taking the spoke and a single iteration of the loop, we can represent a play with a finite lasso.

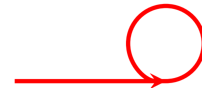

Figure 4.1: A lasso is a path leading into a cycle

We have already shown our Isabelle definitions for lassos in Listing 4.3. A lasso is represented as a set of two lists. The first list forms the spoke that leads to some y, while the second list forms the loop from that y to itself. This representation is useful when we want to argue about these parts separately. For example, we might want to say that a play is won by some player. In this case, we will say that the highest priority in the loop of the lasso is won by that player. The second, alternative definition, `lasso'`, combines the spoke and loop into a single list. We use this definition when we only need to know something about the whole lasso. For instance, we may want to say that the lasso intersects with some region, in which case we can say this for the entire list.

```
lemma origin_in_lasso: "lasso x xs ys ⟹ x ∈ set xs ∨ x ∈ set ys"
  unfolding lasso_def cycle_def
  using path.simps origin_in_path by metis
```

Listing 4.16: Proving the origin of a lasso is part of its paths.

If we represent plays as lassos, we can show properties about them using a standard induction on its list(s). Many of the properties we show for lassos are ones that were initially shown for paths using induction, which were then used to show the same for lassos. For example, we can say that the origin of a lasso is part of one of the two lists using the properties of paths. As shown in Listing 4.16, this is a short automated proof.

4.3 Attractors

By Definition 3.5, an attractor is the set of all nodes for which some player has a strategy that forces any play from those nodes to move to a target region A. In Section 3.1.1, we also give an inductive definition in three steps. We define our attractors and prove their properties in the Isabelle file `Attractors.thy`. Our definitions for attractors in Isabelle are placed in an instance of the `player_paritygame` locale.

The main definition (Listing 4.17) uses the three inductive steps in an inductive set construction. The `base` step adds all nodes in A. The `player` step adds all α-nodes with at least one successor in the attractor. Finally, the `opponent` step adds all $\overline{\alpha}$-nodes that only have successors in the attractor. The `player` and `opponent` cases exclude A, increasing the consistency of when nodes are added and ensuring they are not assigned a strategy in the `player` case.

```
inductive_set player_attractor :: "'v set ⇒ 'v set" for A where
  base: "x ∈ A ⟹ x ∈ player_attractor A"
| player: "⟦x ∈ Vα-A; (x,y) ∈ E; y ∈ player_attractor A⟧
             ⟹ x ∈ player_attractor A"
| opponent: "⟦x ∈ Vα̅-A; ∀y. (x,y) ∈ E ⟶ y ∈ player_attractor A⟧
               ⟹ x ∈ player_attractor A"
```

Listing 4.17: The construction of an attractor in Isabelle.

Initially, our definition for the attractor was a regular inductive definition in the form of a predicate, rather than an inductive set. This was changed because our inductive definition was not exhaustive; it would equally say that A is an attractor to A and that A plus a single α-node with a successor in A is also an attractor to A. To get maximal attractors, further steps would be required. An inductive set automatically constructs the maximal set, making this a simpler solution.

We need to prove that the set obtained with our definition is a valid attractor. Per Definition 3.5, there must exist a σ for α that forces all plays that start in $Attr^{\mathcal{G}}_{\alpha}(A)$ to move to A. This means we need to construct a strategy exhibiting these properties to act as a witness. Our actual definition in Isabelle states that we need a valid strategy that belongs to α; has all α-nodes in the attractor in its domain, except those in A; has a range within the attractor; that the attractor is partially closed in $Attr^{\mathcal{G}}_{\alpha}(A) \setminus A$; and that it leads all plays that start in $Attr^{\mathcal{G}}_{\alpha}(A)$ to A.

In the initial attempt to show these properties, the proof used a structural induction on the steps of the inductive set `player_attractor`, while the strategy had been constructed in its own definition. This proof could not be completed, as Isabelle chooses a successor for α-nodes in the `player` step in the induction. The strategy construction, meanwhile, chose its own successor arbitrarily. There was no way to ensure that these successors were the same in the case that a node had multiple valid successors in the attractor. In another attempt, the strategy was constructed in the induction proof itself. This also could not lead to a finished proof, because we require the strategy to have all of the attractor in its domain. This is not true until the final step of the construction, so it could not be proven for intermediate steps.

The solution that could prove a witness strategy exists uses the concept of ranks, which Zielonka [28] also uses to prove properties of attractors. This concept imagines the construction of an attractor as progressively adding layers to the attractor. These layers are numbered as ranks, with rank 0 being the target A. Each consecutive rank consists of the previous one and adds a layer consisting of all α-nodes with at least one successor in the previous rank, and all $\overline{\alpha}$-nodes that only have successors in the previous rank.

We define ranks as a recursive function `nodes_in_rank`, shown in Listing 4.18. Rank 0 is A. Rank $Suc\, n$, the successor of rank n, is the union of rank n, all α-nodes x where there exists a successor y in rank n, and all $\overline{\alpha}$-nodes x where all successors y of x are in rank n.

```
fun nodes_in_rank :: "nat ⇒ 'v set" where
  "nodes_in_rank 0 = A"
| "nodes_in_rank (Suc n) =
   nodes_in_rank n
   ∪ {x | x y. x ∈ Vα ∧ (x,y) ∈ E ∧ y ∈ nodes_in_rank n}
   ∪ {x. x ∈ Vα̅ ∧ (∀y. (x,y) ∈ E ⟶ y ∈ nodes_in_rank n)}"
```

Listing 4.18: Constructing attractors with ranks.

We prove that this function is monotonic: when $n \leq m$, then rank n is a subset of rank m. We then prove that the `player_tangle_attractor` is equal to a maximum rank. This equality allows us to link properties of the rank construction to our attractor. In particular, we can now show that the rank structure gives us a strategy that forces all plays into A, by using the property that we can only move one step closer to A from any node in a rank.

Lemma 4.2. *For all ranks n for α, there exists a σ such that:*

1. *σ is a valid strategy of α*
2. *the domain of σ contains all α nodes in rank n, except for those in A*
3. *the range of σ is a subset of rank n*
4. *all successors of nodes in ranks lower or equal to n, except for those in A, move to the same rank, and*
5. *from all nodes x in rank n, all paths longer than n steps intersect A.*

Proof. In our Isabelle proof, we do an induction on n. If we have $n = 0$, then an empty strategy suffices. Property 1 is satisfied because empty strategies can be valid for either player. Property 2 excludes nodes in A, so it is trivially satisfied. Property 3 is satisfied because an empty range is a subset of the rank n. Property 4 once again excludes nodes in A, meaning it is trivially satisfied. Finally, any path that starts in A is already in A, satisfying property 5.

```
lemma nodes_in_rank_forces_A: "∃σ.
  strategy_of Vα σ ∧
  dom σ = (nodes_in_rank n-A) ∩ Vα ∧
  ran σ ⊆ nodes_in_rank n ∧
  (∀n'. n'≤n ⟶ (∀x' ∈ nodes_in_rank n'-A. ∀y ∈ (induced_subgraph σ) `` {x'}.
    y' ∈ nodes_in_rank n')) ∧
  (∀x ∈ nodes_in_rank n. ∀xs z. path (induced_subgraph σ) x xs z ∧ n<length xs
    ⟶ set xs ∩ A ≠ {})"
proof (induction n)
  case 0 thus ?case
    apply (rule exI[where x=Map.empty])
    using origin_in_path by fastforce
```

For a rank *Suc* n, the successor of some rank n, we know by our induction hypothesis that we have a σ that satisfies our properties for rank n. We obtain this strategy and give shorthand for its subgame $\mathcal{G}[\sigma]$.

```
next
  case (Suc n)
  from Suc.IH obtain σ where
    strat_σ: "strategy_of Vα σ" and
    dom_σ: "dom σ = (nodes_in_rank n-A) ∩ Vα" and
    ran_σ: "ran σ ⊆ nodes_in_rank n" and
    closed_σ: "∀n'. n'≤n ⟶ (∀x' ∈ nodes_in_rank n'-A.
      ∀y' ∈ (induced_subgraph σ) `` {x'}. y' ∈ nodes_in_rank n')" and
    forces_σ:
      "∀x∈nodes_in_rank n. ∀xs z. path (induced_subgraph σ) x xs z ∧ n<length xs
        ⟶ set xs ∩ A ≠ {}"
    by blast
  let ?Gσ = "induced_subgraph σ"
```

We extend this strategy by picking an arbitrary valid successor for any α nodes that were newly added in rank *Suc* n. This requires some definitions. We first give a definition for all newly added α-nodes. Next, we define a function, `target`, that picks a successor in the previous rank n for any node it is given. This is defined as a lambda expression, and uses the Hilbert choice operator `SOME` to obtain an arbitrary valid successor in rank n of a given x.

```
  define new_player_nodes where
    "new_player_nodes = (nodes_in_rank (Suc n) - nodes_in_rank n) ∩ Vα"
  define target where
    "target ≡ λx. SOME x'. x' ∈ nodes_in_rank n ∧ (x,x') ∈ E"
```

We show that, if a node is one of the `new_player_nodes`, it is a node in the current rank *Suc* n, it is a α-node, it is not part of the previous rank n, the successor selected for it by `target` is a node in rank n, and that node is a valid successor. This will be useful for later properties.

```
have target_eq: "x ∈ new_player_nodes ⟷
  (x ∈ nodes_in_rank (Suc n) ∧
   x ∈ V_α ∧
   x ∉ nodes_in_rank n ∧
   target x ∈ nodes_in_rank n ∧
   (x,target x) ∈ E)" for x
  apply (rule iffI; simp add: new_player_nodes_def)
  using some_eq_imp[of _ "target X"]
  unfolding target_def by blast+
```

Now, we define the strategy σ' as a lambda expression. If a node is one of the newly added α-nodes, it is assigned a valid successor in rank n with `target`. Otherwise, its successor is chosen by the original σ. We also give shorthand for $\mathcal{G}[\sigma']$. Now, we have to show that this is our valid witness strategy.

```
define σ' where
  "σ' ≡ λx. if x ∈ new_player_nodes then Some (target x) else σ x"
let ?Gσ' = "induced_subgraph σ'"

show ?case
proof (rule exI[where x=σ']; intro conjI allI ballI impI; (elim conjE)?)
```

The first property, that σ' is a valid strategy for α, is satisfied because we extend the original σ, which was already a valid strategy, with one that assigns valid successors to α-nodes. This means the combined strategy is also a valid strategy for α.

```
  from strat_σ show σ'_strat: "strategy_of V_α σ"
    unfolding σ'_def strategy_of_def E_of_strat_def
    by (safe; simp add: target_eq split: if_splits) blast+
```

The second property states that the domain of σ' contains all α-nodes in rank $Suc\ n$, except those in A. It is true because our original σ already had all the required nodes in rank n in its domain. `target` is used to assign one to every new α-node in the rank $Suc\ n$, so the combined strategy has a move for every α-node in the rank $Suc\ n$, excluding those in A.

```
  from dom_σ show σ'_dom: "dom σ' = (nodes_in_rank (Suc n)-A) ∩ V_α"
    unfolding σ'_def new_player_nodes_def
    apply (safe; simp split: if_splits)
    using nodes_in_rank_subset by blast+
```

The third property requires the range of σ' to be a subset of the rank $Suc\ n$. The original σ had a subset of the rank n as its range. `target` only picks nodes in that same rank. Since this rank is a subset of the rank $Suc\ n$, the range of the combined strategy is a subset of that rank.

```
  from ran_σ show σ'_ran: "ran σ' ⊆ nodes_in_rank (Suc n)"
    unfolding σ'_def by (clarsimp simp: ran_def target_eq) blast
```

The fourth property states that all lower ranks n' need to have their rank be partially closed in `nodes_in_rank n' - A`. We prove this this within its own separate context, where we fix our rank n', a node x', and its successor y'. We assume that n' is less than or equal to $Suc\ n$, that it is a node in the rank n', but not in A, and that y' is its successor in $\mathcal{G}[\sigma']$. We then look at two different cases: one where $n' \leq n$, and one where $n' = Suc\ n$.

```
  {
    fix n' x' y'
    assume n'_leq_Suc_n: "n' ≤ Suc n"
       and x'_in_n'_min_A: "x' ∈ nodes_in_rank n'-A"
       and y'_succ_X: "y' ∈ ?Gσ' `` {x'}"

    then consider (n'_leq_n) "n'≤n" | (n'_Suc_n) "n'=Suc n" by linarith
    thus "y' ∈ nodes_in_rank n'" proof cases
```

In the case where $n' \leq n$, our x' is not part of the rank $Suc\, n$, so it is not one of the newly added α-nodes. This means the edge from x' to y' exists in $\mathcal{G}[\sigma]$ of our original σ. The ranks equal to and below n are partially closed in that rank minus A in $\mathcal{G}[\sigma]$, so this edge leads to the same rank. This shows the thesis holds in this case.

```
    case n'_leq_n
    from nodes_in_rank_mono[OF this] x'_in_n'_min_A
    have "x' ∉ new_player_nodes"
      unfolding new_player_nodes_def by blast

    with y'_succ_x' have "(x',y') ∈ ?Gσ"
      unfolding σ'_def induced_subgraph_def E_of_strat_def
      by auto

    with x'_in_n'_min_A n'_leq_n closed_σ show ?thesis by blast
```

In the case where $n' = Suc\, n$, we know that x' is part of the rank $Suc\, n$, and it lies outside of A. If it is a α-node, this means it lies in the domain of σ'. Therefore, y' must then lie in the range of σ'. The range of σ' is a subset of the rank $Suc\, n$, so the thesis holds if that is the case. If x' is not a α-node, we can use an auxiliary lemma that shows that $\overline{\alpha}$ nodes in a rank have successors in that same rank. This shows that in that case, y' is also a node in the rank $Suc\, n$. Therefore, the thesis holds for that case as well. This completes the proof of the partial closedness of the ranks equal to and below $Suc\, n$.

```
  next
    case n'_Suc_n
    with x'_in_n'_min_A y'_succ_x' have
      x'_in_suc: "x' ∈ nodes_in_rank (Suc n)" and
      x'_notin_A: "x' ∉ A" and
      edge: "(x',y') ∈ E" by auto

    with y'_succ_x' σ'_dom σ'_ran show ?thesis
      unfolding induced_subgraph_def E_of_strat_def
      apply (cases "x ∈ V_α"; simp add: <n'=Suc n> target_eq ran_def)
      using nodes_in_rank_edges_same[OF x'_in_suc <x'∉A> edge] by auto
  qed
} note closed_σ'=this
```

The final property requires all paths longer than the length $Suc\, n$ intersect with A. To make this easier to prove, we first define an auxiliary lemma to show that any path starting from rank n that does not intersect with A also exists in $\mathcal{G}[\sigma]$ of the original σ. We begin by fixing our path in $\mathcal{G}[\sigma']$ from a node in rank n, and assume it does not intersect with A.

```
{
  fix x xs y
  assume x_in_n: "x ∈ nodes_in_rank n"
     and path: "path ?Gσ' x xs y"
     and xs_no_A: "A ∩ set xs = {}"
```

We prove that this path also exists in $\mathcal{G}[\sigma]$ by considering the possible cases of xs. If it is an empty path, the thesis is trivially true, as empty paths always exist in any subgame.

```
  have "path ?Gσ x xs y"
  proof (cases xs)
    case Nil thus ?thesis using path by auto
```

In the case where xs consists of a head and a tail, it is not empty. This means the first element is x, and since xs does not intersect with A, x is also not a node in A. We state then that x is part of rank n with A removed.

```
  next
    case (Cons a list)
    with xs_no_A have "x ∉ A"
      using origin_in_path[OF path] by blast
    with x_in_n have x_in_n_min_A: "x ∈ nodes_in_rank n - A" by blast
```

We state that $\mathcal{G}[\sigma']$ is a subgraph of $\mathcal{G}[\sigma]$ within the rank n. This is true because the moves of σ are the same within this region.

```
    have subgraph:
      "Restr (induced_subgraph σ') (nodes_in_rank n) ⊆ induced_subgraph σ"
```

We also rephrase the closedness of rank n in a way that lets us use it directly in an upcoming lemma.

```
    from closed_σ'[of n] have
      "?Gσ' `` (nodes_in_rank n-A) ⊆ nodes_in_rank n"
      by auto
```

Now, we can see that the path starts in rank n and never intersects with A, which means it will stay in rank n. By our subgraph property, this means the path also exists in $\mathcal{G}[\sigma]$, completing this auxiliary lemma. We name this lemma `xfer_lower_rank_path`.

```
    from path_partially_closed[OF x_in_n_min_A this path] xs_no_A have
      "set xs ⊆ nodes_in_rank n" "y ∈ nodes_in_rank n" by blast+
    from subgraph_path[OF subgraph path_restr_V[OF path this]]
    show ?thesis.
  qed
} note xfer_lower_rank_path=this
```

We start proving the final property. We do this in a context where we fix a node x in the rank $Suc\, n$, and a path from it with a length longer than $Suc\, n$ steps. We can make a case distinction on this node x. It can either be a node that was already in rank n, it can be a newly added α-node, or it can be a newly added $\overline{\alpha}$-node.

```
{
  fix x xs z
  assume x_in_suc: "x ∈ nodes_in_rank (Suc n)"
     and path: "path ?Gσ' x xs z"
     and len: "Suc n < length xs"

  from x_in_suc consider
    (already_in) "x∈nodes_in_rank n"
  | (our_node) "x∉nodes_in_rank n" "x∈V_α" "(x,target x)∈E"
      "target x∈nodes_in_rank n"
  | (opponent_node) "x∉nodes_in_rank n" "x∈V_ᾱ"
      "∀y∈E``{x}. y∈nodes_in_rank n"
    using new_player_nodes_def target_eq by simp blast
  thus "set xs ∩ A ≠ {}" proof cases
```

In the case that x is a node that was already part of the rank n, we know the whole path stays in that previous rank. By our auxiliary lemma, and the fact that the original σ forces plays to A, this path does the same.

```
    case already_in with path len forces_σ show ?thesis
      using xfer_lower_rank_path Suc_lessD by fast
```

In the case that x is a newly added α-node, we know its successor is one in rank n, selected with `target`. We get the path xs' from that node to z.

```
  next
    case our_node
    with x_in_suc have "(x,y)∈?Gσ' ⟹ y=target x" for y
      unfolding σ'_def induced_subgraph_def E_of_strat_def
      by (auto simp: target_eq split: if_splits)
```

```
    with path len obtain xs' where
      xs': "xs=x#xs" "path ?Gσ' (target x) xs' z"
      by (cases xs) auto
```

Now, we show that xs intersects with A using a proof by contradiction. We assume that it does not intersect with A. This means that the path xs' now exists entirely within the original $\mathcal{G}[\sigma]$ by our auxiliary lemma. Because σ forces all paths longer than n steps to intersect with A, this is also true for this path. This contradicts our assumption that it does not intersect with A, completing the proof.

```
    show "set xs ∩ A ≠ {}" proof
      assume xs_no_A: "set xs ∩ A = {}"
      with xfer_lower_rank_rank_path[OF <target x ∈ nodes_in_rank n] xs'
      have "path ?Gσ (target x) xs' z" by auto
      with xs_no_A len forces_σ <target x ∈ nodes_in_rank n> xs'
      show False by auto
    qed
```

In the case that x is a newly added $\overline{\alpha}$-node, we follow a similar structure. All successors of this node must be nodes in the rank n, so we obtain the path xs' from the successor of x to z. We do another proof by contradiction, where the path xs' must exist in $\mathcal{G}[\sigma]$, thus it is forced to intersect with A.

```
  next
    case opponent_node
    with len path obtain xs' y where
      xs': "xs=x#xs" "path ?Gσ' y xs' z" "y ∈ nodes_in_rank n"
      by (cases xs) auto

    show "set xs ∩ A ≠ {}" proof
      assume xs_no_A: "set xs ∩ A = {}"
      with xfer_lower_rank_path xs'
      have "path ?Gσ y xs' z" by auto
      with xs_no_A len forces_σ xs'
      show False by auto
    qed
  qed
} note σ'_forces_A=this
qed
qed
```

This completes our proof showing that each rank has a strategy σ that satisfies the properties outlined in Lemma 4.2. ☐

Having proven a witness strategy exists for the rank structure, we can now prove that the strategy also exists for the inductive set. To do so, we show that there exists a maximum rank that is equivalent to the inductive set. Then, we take the strategy for this rank and use this as our witness for the inductive set.

4.4 Proof of Positional Determinancy

Now, we show the positional determinancy of parity games using our definitions. If our definitions are correct, positional determinancy should hold, hence we use this proof to check the correctness of our definitions. We show positional determinancy in the file `PositionalDeterminancy.thy`. After some additional lemmas to help in our proof, we show the following lemma:

Lemma 4.3. *Every parity game $\mathcal{G}$ can be partitioned into two disjoint winning regions W_1 and W_2 for players Even and Odd, respectively.*

Proof. Our Isabelle proof is adapted from Zielonka's proof [28]. The proof is over 500 lines in Isabelle, so we will condense most of it for readability as we follow along.

The proof uses an induction on the size of the game. In Isabelle, we use the induction rule `finite_psubset_induct`, which lets us use an induction on a finite set. We get a case where our induction hypothesis operates on strict subsets of the current set. We must first show that our V is finite, before we can start the actual induction.

```
lemma maximal_winning_regions:
  fixes V :: "'v set"
  assumes "paritygame E V V0"
  shows "∃W0 W1. V = W0 ∪ W1 ∧ W0 ∩ W1 = {}
    ∧ paritygame.winning_region E V V0 pr EVEN W0
    ∧ paritygame.winning_region E V V0 pr ODD W1"
proof -
  have "finite V" proof -
    interpret paritygame E V V0 by fact
    show ?thesis by blast
  qed

  thus ?thesis using assms
  proof (induction arbitrary: E V0 rule: finite_psubset_induct)
    case (psubset V)
    interpret paritygame E V V0 by fact
```

We consider two cases here: one where the game is empty, and one where it is not empty. In the former case, we trivially have two winning regions, which are both empty.

```
    consider (V_empty) "V = {}" | (V_notempty) "V ≠ {}" by fast
    thus ?case proof cases
      case V_empty thus ?thesis
        using winning_region_empty by auto
```

In the case that the game is not empty, we start by getting the highest priority p in the game, and the player $\alpha \equiv_2 p$ who wins p. We also define our shorthand for V_α, $\overline{\alpha}$, and $V_{\overline{\alpha}}$ with `let`.

```
    next
      case V_notempty

      define p where "p = pr_set V"

      then obtain α where "α = player_wins_pr p" by simp
      hence player_wins_p: "player_winningP α p"
        by (cases α; simp add: player_wins_pr_def split: if_splits)
      let ?Vα = "V_player α"
      let ?α̅ = "opponent α"
      let ?Vα̅ = "V_opponent α"
```

We attract to v for α and call the attracted region A. Removing an attractor from a game gives a valid subgame, so $V \setminus A$ is now a valid subgame.

```
      define A where "A = attractor α {v}"
      with <v∈V> have "V ∈ A" using attractor_subset by blast

      interpret subgame: paritygame "Restr E (V-A)" "V-A" "V0-A"
        using attractor_subgame[OF A_def] by blast
```

We show that, because A contains v, $V \setminus A$ is a strict subset of V. Now, by our induction hypothesis, we have two winning regions in $V \setminus A$: W_α and $W_{\overline{\alpha}}$ for α and $\overline{\alpha}$, respectively.

```
      from <v∈A> <v∈V> have "V-A ⊂ V" by blast
      from psubset.IH[OF this subgame.paritygame_axioms]
      obtain Wα Wα̅ where
        W_comp: "V-A = Wα ∪ Wα̅" and
        W_disj: "Wα ∩ Wα̅ = {}" and
```

```
    Wα_win: "subgame.winning_region α Wα" and
    Wα̅_win: "subgame.winning_region α̅ Wα̅"
    using opponent.simps(2) apply (cases α)
    by fastforce (metis Int_commute Un_commute)
```

We once again have two possibilities: one where $W_{\overline{\alpha}}$ is empty, and one where it is not.

```
  consider (Wα̅_empty) "Wα̅ = {}" | (Wα̅_notempty) "Wα̅ ≠ {}" by blast
  thus ?thesis proof cases
```

If $W_{\overline{\alpha}}$ is empty, our graph consists entirely of $W_\alpha \cup A$.

```
    case Wα̅_empty
    with <v∈V> W_comp A_def have V_comp: "V = Wα ∪ A"
      using attractor_subset_graph[of "{v}"] by blast
```

Furthermore, this region is now the winning region for α. The region W_α is the same in the whole graph as it is in $V \setminus A$. Under the winning strategy σ for W_α, all plays with their loop in W_α are won by α in $V \setminus A$, which is maintained in V. Any play which has A in its loop is forced to contain v in that loop, using the attractor strategy σ' for A. Because v has the highest priority in the game, it is also the highest priority in the loop. The priority of v is won by α, so this play is also won by α. This means that these two strategies combined almost make up the winning strategy for α in $W_\alpha \cup A$. If the node v belongs to α, then we must assign a valid successor to it in the combined strategy to ensure the strategy is complete. Which successor we pick does not matter, as the two regions make up the whole graph.

```
    moreover have "winning_region α (Wα∪A)"
    proof -
      ...
    qed
```

Since this makes up the whole graph, the winning region for $\overline{\alpha}$ is an empty region. This completes the proof that we have two winning regions for this case.

```
    ultimately show ?thesis
      using winning_region_empty[of ?α̅] by (cases α; auto)
```

If $W_{\overline{\alpha}}$ is not empty, we attract for $\overline{\alpha}$ to this region. Once again, this makes $V \setminus B$ a valid subgame, since B is an attractor.

```
  next
    case Wα̅_notempty

    define B where "B = attractor α̅ Wα̅"
    with Wα̅_notempty W_comp have "B ≠ {}" "B ⊆ V"
      using attractor_subset attractor_subset_graph by blast+

    interpret subgame': paritygame "Restr E (V-B)" "V-B" "V0-B"
      using attractor_subgame[OF B_def] by blast
```

This means we can use our induction hypothesis again. We have two winning regions in $V \setminus B$: X_α and $X_{\overline{\alpha}}$ for α and $\overline{\alpha}$, respectively.

```
    from <B≠{}> <B⊆V> have "V-B ⊂ V" by blast
    from psubset.IH[OF this subgame'.paritygame_axioms]
    obtain Xα Xα̅ where
      X_comp: "V-B = Xα ∪ Xα̅" and
      X_disj: "Xα ∩ Xα̅ = {}" and
      Xα_win: "subgame'.winning_region α Xα" and
      Xα̅_win: "subgame'.winning_region α̅ Xα̅"
      using opponent.simps(2) apply (cases α)
      by fastforce (metis Int_commute Un_commute)
```

Now, the whole graph consists of the disjoint regions X_α and $X_{\overline{\alpha}} \cup B$.

```
from X_comp <B⊆V> have V_comp: "V = Xα ∪ (Xα̅∪B)"

moreover from X_comp X_disj have disj: "Xα ∩ (Xα̅∪B) = {}"
```

Furthermore, X_α is a winning region for α in the whole game. When we remove an attractor for some player from the game, then any winning region for that player's opponent in the resulting subgame is also a winning region in the whole game. The strategy for that region makes all nodes belonging to its winner move to another node in the region, and any node belonging to its loser that could have moved to their own winning region would have done so in the subgame, while any node belonging to the loser that could have moved to the attractor in the whole game would have been attracted in the first place.

```
moreover from Xα_win B_def have "winning_region α Xα"
using attractor_subgame_winning_region[OF subgame'.paritygame_axioms]
using subgame'.winning_region_in_V by force
```

Moreover, the region $X_\alpha \cup B$ is the winning region for $\overline{\alpha}$. $W_{\overline{\alpha}}$ is a winning region for $\overline{\alpha}$ in the subgraph $V \setminus A$, and because A is an attractor for α, $W_{\overline{\alpha}}$ is also a winning region for $\overline{\alpha}$ in V. We can extend any winning region by attracting to it for the winning player, so B is now also a winning region for $\overline{\alpha}$ in V. Now, any play with its loop in $X_{\overline{\alpha}}$ will be won by $\overline{\alpha}$ in V, as the game in this region is the same in V and $V \setminus A$. Any play that has B in its loop will stay entirely in B, as B was already a closed winning region in V, and therefore is also won by $\overline{\alpha}$. Therefore, the combined region is winning for $\overline{\alpha}$ with the winning strategy being the combined winning strategies of $X_{\overline{\alpha}}$ and B.

```
moreover have "winning_region α̅ (Xα̅∪B)"
proof -
  ...
qed
```

Using the aforementioned facts, we prove our thesis for this case.

```
  ultimately show ?thesis
    by (cases α; simp) blast+
qed
...
```

With this, we have completed our proof that each graph can be split into two disjoint winning regions. □

While this pen-and-paper proof is relatively straightforward, proving this in Isabelle introduces complications. A common source of complexity is having to prove that some strategy from a subgame, or some combination of strategies, is a winning strategy. If the strategy originates from a subgame, we need to show that its properties hold in the whole game. If it is a combination of two existing strategies, we must show that their combined properties exhibit the required properties of a winning strategy. This usually consists of one step for each independent property of a winning strategy, quickly bloating the size of the proof.

In various instances, we have a strategy that is winning for α in a subgame, and must show for that strategy that all paths in the whole game are won by α. The structure for these proofs is quite similar each time. We have a lasso in the whole game, and must show that the lasso (or at least its loop) stays in the same region as the winning region in the subgame. Then, we show that this region is identical in the subgame, so the cycle of this lasso also exists in the subgame. Since this region is winning in the subgame, the cycle is won by the player, thus showing that this play is also won in the whole game.

We also mentioned we needed additional lemmas. These are used in the proof of Lemma 4.3. For instance, we must show that removing an attractor from a game gives a valid subgame. We must also show that an attractor can be used to extend a winning region. Another example is a lemma that shows that if a subgame was obtained by removing an attractor for α, then a winning region for α in the subgame is still a winning region in the whole game.

With the proof for Lemma 4.3 completed, we can show positional determinancy. The relevant proofs are shown in Listing 4.19. We start by showing that all nodes $v \in V$ are won by a player. We use

Lemma 4.3, as well as an earlier lemma (not shown in this thesis) that states every node in a winning region for α is won by α. Next, we show that a node cannot be won by both players using Lemma 4.1. Finally, we give a single statement that shows positional determinancy, `v_positionally_determined`. It states that if v is in V, then the values of `won_by EVEN` and `won_by ODD` are not equal. These are Boolean values, so one must always be `True`, while the other must always be `False`.

```
lemma all_v_won:
  assumes: "v ∈ V"
  shows: "won_by EVEN v ∨ won_by ODD v"
  using assms maximal_winning_regions[OF paritygame_axioms] winning_region_won_by
  by blast

lemma v_won_by_one_player: "¬(won_by EVEN v ∧ won_by ODD v)"
  using won_by_player_not_won_by_opponent by fastforce

theorem v_positionally_determined:
  assumes "v ∈ V"
  shows "won_by EVEN v ≠ won_by ODD v"
  using all_v_won[OF <v∈V>] v_won_by_one_player by blast
```

Listing 4.19: Proving that the winner of each v is determined.

With this final statement, we have shown that every v in V must always be won by one player and never by both.

Chapter 5

Formalisation of Tangle Learning

With the formalisation of parity games completed, we can move on to tangle learning itself. We build our definitions on our formalisation of parity games and prove relevant properties of our definitions, before giving our correctness proof for the tangle learning algorithm. First, we define tangles in Section 5.1. Next, in Section 5.2, we define tangle attractors and prove the correctness of this definition. Section 5.3 then gives our formalisation of the `search` algorithm, as well as its correctness proof. Finally, we discuss how the `solve` algorithm may be formalised in Section 5.4.

5.1 Tangles

Tangles form the basis of the tangle learning algorithm. They are defined in the file `tangles.thy`. Recall Definition 3.7: A tangle is a nonempty set of nodes $U \subseteq V$ if α has a strategy $\sigma : U_\alpha \to U$, such that the graph (U, E'), with $E' := E \cap (\sigma \cup U_{\overline{\alpha}} \times U))$, is strongly connected, and α wins all cycles in (U, E').

We define tangles first in the `player_paritygame` locale. Our definition is split up into reusable parts, starting with the tangle subgraph (U, E'). Our definition in Isabelle (Listing 5.1) closely follows the original definition. We intersect E with the edges of σ and the $\overline{\alpha}$-nodes in U.

```
definition player_tangle_subgraph :: "'v set ⇒ 'v strat ⇒ 'v dgraph" where
  "player_tangle_subgraph U σ ≡ E ∩ (E_of_strat σ ∪ ((U ∩ V_α̅) × U)"
```

Listing 5.1: A player's tangle subgraph (U, E').

This definition closely resembles a regular induced subgraph $\mathcal{G}[\sigma]$. We find that it is actually the same as $\mathcal{G}[\sigma]$, restricted to the region U. Because we work with induced subgraphs most often, we define a lemma (shown in Listing 5.2) that expresses this relation, allowing us to use it in our future proofs.

```
lemma player_tangle_subgraph_is_restr_ind_subgraph:
  assumes "U ⊆ V"
  assumes "dom σ = U ∩ V"
  assumes "ran σ ⊆ U"
  shows "player_tangle_subgraph U σ = Restr (induced_subgraph σ) U"
  unfolding player_tangle_subgraph_def induced_subgraph_def E_of_strat_def
  using assms by (auto simp: ranI)
```

Listing 5.2: A tangle subgraph for a player is that same as restricting the induced subgraph of its strategy to the tangle.

Next, we define the strategy for the tangle. This must be a strategy belonging to α which covers all α-nodes in U, has a range in U, and (U, E') is strongly connected and all cycles in (U, E') are won by the player:

```
definition player_tangle_strat :: "'v set ⇒ 'v strat ⇒ bool" where
  "player_tangle_strat U σ ≡
    strategy_of V_α σ ∧ dom σ = U ∩ V_α ∧ ran σ ⊆ U ∧
    (let E' = player_tangle_subgraph U σ in (
      strongly_connected E' U ∧
      (∀v ∈ U. ∀xs. cycle E' v xs ⟶ winning_player xs)
    ))"
```

Listing 5.3: A tangle's strategy for a player.

Note the use of the `let` keyword here. Like in an isar proof, it lets us set named variables that we can reuse for readability and to make definitions more succinct. In this case, every instance of E' within the brackets after `in` is the tangle subgraph.

Finally, we define a tangle for a player as `player_tangle`. A tangle is a nonempty subset U of all nodes V, where the highest priority in U is winning for the player, and there exists a strategy σ that is a tangle strategy for U. To take this into the locale `paritygame`, we once again use a function that references this definition. Both definitions can be seen in Listing 5.4.

```
definition player_tangle :: "'v set ⇒ bool" where
  "player_tangle U ≡
    U ≠ {} ∧ U ⊆ V ∧ winningP (pr_set U) ∧ (∃σ. player_tangle_strat U σ)"

fun tangle :: "player ⇒ 'v set ⇒ bool" where
  "tangle EVEN = P0.player_tangle"
| "tangle ODD = P1.player_tangle"
```

Listing 5.4: Defining tangles.

We also prove a property of tangles that is stated in Van Dijk's seventh lemma for the correctness of tangle learning [7, Section 4.5]:

Lemma 5.1. *A tangle t is a winning region if and only if $E_T(t) \neq \emptyset$.*

Our Isabelle proof (Listing 5.5) for this lemma depends on two auxiliary lemmas: one that states that a region without escapes is closed for the opponent, and one that a closed tangle is a winning region (one of the observations noted in Section 3.1.3).

```
lemma no_opponent_escapes_tangle_is_winning_region:
  assumes "player_tangle t"
  shows "opponent_escapes t = {} ⟷ player_winning_region t"
  using assms
  using no_escapes_closed_opponent[of t]
  using closed_player_tangle_is_winning_region[of t]
  by (auto simp: player_winning_region_def)

lemma no_escapes_tangle_is_winning_region:
  assumes "tangle α t"
  shows "escapes α t = {} ⟷ winning_region α t"
  using assms
  using P0.no_opponent_escapes_tangle_is_winning_region[of t]
  using P1.no_opponent_escapes_tangle_is_winning_region[of t]
  by (cases α; simp)
```

Listing 5.5: Our Isabelle proofs for Lemma 5.1.

5.2 Tangle Attractors

The way we formalise tangle attractors is different from how we defined standard attractors. We include the construction of the witness strategy alongside the construction of the attractor set. Consequently, we cannot use an inductive set anymore: it cannot construct both a set and a strategy at the same time. We have to return to an inductive definition. This has a `player` step for adding α-nodes that have at least one successor in the attractor, an `opponent`-step for adding $\overline{\alpha}$-nodes that only have successors in the attractor, and a `tangle` step for adding tangles for α from which $\overline{\alpha}$ can only escape into the attractor. We open a context that fixes a finite set of known tangles for α, which we name T. We also fix our target set A. This allows us work with them in our definitions without having to explicitly include them as parameters. Within these contexts, we define our attractor construction as a step relation:

```
inductive
  tangle_attractor_step :: "'v set × 'v strat ⇒ 'v set × 'v strat ⇒ bool"
where
  player: "⟦x ∈ V_α-X; (x,y) ∈ E; y ∈ X⟧
    ⟹ tangle_attractor_step (X,σ) (insert x X,σ(x↦y))"
| opponent: "⟦ x ∈ V_ᾱ-X; ∀y. (x,y) ∈ E ⟶ y ∈ X⟧
    ⟹ tangle_attractor_step (X,σ) (insert x X,σ)"
| tangle: "⟦t ∈ T; t-X ≠ {}; opponent_escapes t ≠ {}; opponent_escapes t ⊆ X;
    player_tangle_strat t σ'⟧
    ⟹ tangle_attractor_step (X,σ) (X ∪ t, (σ' |` (t-A)) ++ σ)"
```

Listing 5.6: An inductive definition for constructing tangle attractors.

The automatically generated induction rule for our step does not work when we explicitly specify X, σ, X', and σ' in a lemma. Instead, it requires the pairs be specified as single variables. This is inconvenient because having to work with pairs makes lemmas and proofs harder to read. Therefore, we define our own induction rule, `tangle_attractor_step_induct`, which works the same as the usual induction rule, while letting us specify the variables in the pairs individually. We will use this rule in further proofs.

In Section 4.3 we mentioned that the inductive set automatically takes inductive steps until no more can be taken, giving us a maximal attractor. Because our new definition is an inductive predicate for the step relation instead, we have to include maximality in our definition. Our definition for the a player's tangle attractor uses the reflexive transitive closure of our step relation to say that we can reach the final state from the initial state, and we say that the final state cannot be in the domain of the relation, allowing no further steps to be taken from it.

Besides this, we need to add one more element to our definition: we need to pick a successor in the attractor X for every α-node in A where this is possible. For this, we define `A_target` (Listing 5.7), which takes X as input and gives us a strategy (in the form of a lambda expression) that, if a valid candidate exists, picks any arbitrary successor in X for an α-node x in A using the Hilbert choice operator `SOME`, and otherwise gives us nothing.

```
definition A_target :: "'v set ⇒ 'v strat" where
  "A_target X ≡ λx.
   if x ∈ V_α ∩ A ∧ (∃y. y ∈ X ∧ (x,y) ∈ E)
   then Some (SOME y. y ∈ X ∧ (x,y) ∈ E)
   else None"
```

Listing 5.7: A function for picking successors in X for α-nodes in A.

We prove that `A_target` is a valid strategy for the player. Furthermore, we prove that the domain of `A_target` is some subset of all α-nodes in A, and that its range lies within X. Finally, we prove that, if the domain of a strategy σ does not contain any part of A, then the subgame of that strategy combined with `A_target X` is a subgraph of $\mathcal{G}[\sigma]$, and $\mathcal{G}[\sigma]$ is a subgraph of the combined strategy outside of A. We will use these lemmas when we prove the properties of our attractor and its strategy. In Isabelle, these are short automated proofs, shown in Listing 5.8.

```
lemma A_target_strat: "strategy_of Vα (A_target X)"
  unfolding A_target_def strategy_of_def E_of_strat_def
  apply (rule conjI; clarsimp split: if_splits)
  using someI2[of "λy. y ∈ X ∧ (x,y) ∈ E" for x] by fast

lemma A_target_dom: "dom (A_target X) ⊆ Vα ∩ A"
  unfolding A_target_def by (auto split: if_splits)

lemma A_target_ran: "ran (A_target X) ⊆ Vα ∩ A"
  apply (clarsimp simp: A_target_def ran_def)
  using someI_ex[of "λy. y ∈ X ∧ (x,y) ∈ E" for x] by blast

lemma add_A_target_A_notin_dom:
  assumes "dom σ ∩ A = {}"
  shows "induced_subgraph (σ ++ A_target X) ⊆ induced_subgraph σ"
    and "Restr (induced_subgraph σ) (-A) ⊆ induced_subgraph (σ ++ A_target X)"
  unfolding induced_subgraph_def E_of_strat_def A_target_def
  using assms by (auto split: if_splits simp: map_add_Some_iff)
```

Listing 5.8: Additional properties of A_target.

Now, we can finally construct our definition for the tangle attractor (Listing 5.9). We say that a set X and a strategy σ are a valid tangle attractor and its strategy if there exists a σ' such that the pair (X, σ') is obtained by exhaustively taking steps from the target A, and σ is constructed by adding the edges from α-nodes in A to X using `A_target`.

```
definition player_tangle_attractor :: "'v set ⇒ 'v strat → bool" where
  "player_tangle_attractor X σ ≡ ∃σ'.
     tangle_attractor_step** (A,Map.empty) (X,σ')
   ∧ ¬Domainp tangle_attractor_step (X,σ')
   ∧ σ = σ' ++ A_target X"
```

Listing 5.9: Our definition of a tangle attractor for a specific player.

Since we fixed the set of known tangles in the context our tangle attractor resides in, when we place attractors in the `paritygame` locale (Listing 5.10), we need to filter the set of all known tangles to get only those tangles that belong to the relevant player. This way, we do not to perform this filtering when we use tangle attractors in the tangle learning algorithm.

```
fun tangle_attractor
  :: "player ⇒ 'v set set ⇒ 'v set ⇒ 'v set ⇒ 'v strat ⇒ bool"
where
  "tangle_attractor EVEN T = P0.player_tangle_attractor {t∈T. tangle EVEN t}"
| "tangle_attractor ODD T = P1.player_tangle_attractor {t∈T. tangle ODD t}"
```

Listing 5.10: Tangle attractors in the paritygame locale.

5.2.1 Correctness Proof

We will show the correctness of our definition. To do so, we must show that the result of our algorithm exhibits the desired properties. We need to show that the obtained strategy σ is a valid witness strategy for the obtained region X being a tangle attractor. We also include in our correctness the property that there always exists a path to A under σ in our attractor. This is lemma 3 in Van Dijk's pen-and-paper proof [7, Section 4.5], and we will need it for the correctness proof of `search` later.

Properties of Our Step Relation

The aforementioned properties are encoded in an invariant. We will have to show that this invariant holds at the base step, when X is A and σ is empty, and then show that each step preserves the properties of the invariant. Our invariant (Listing 5.11) is defined as a lambda expression that takes

a state (X, σ) as input. This invariant holds if A is a subset of X, σ is a valid strategy for α, the domain of σ is all of $V_\alpha \cap (X \setminus A)$, the range of σ is within X, X is partially closed in $X \setminus A$ in $\mathcal{G}[\sigma]$, all plays starting in X either move to A or are won by α, and there exists a path to A from any node in X.

```
definition tangle_attractor_step_I :: "'v set × 'v strat ⇒ bool" where
  "tangle_attractor_step_I ≡ λ(X,σ).
   A ⊆ X ∧
   strategy_of Vα σ ∧
   dom σ = Vα ∩ (X-A) ∧
   ran σ ⊆ X ∧
   induced_subgraph σ `` (X-A) ⊆ X ∧
   (∀x∈X. ∀xs ys. lasso (induced_subgraph σ) x xs ys
    ⟶ (set (xs@ys) ∩ A ≠ {} ∨ winning_player ys)) ∧
   (∀x∈X. ∃x'∈A. ∃xs. path (induced_subgraph σ) x xs x')"
```

Listing 5.11: Our invariant for the tangle attractor step.

We will now prove relevant properties of the tangle attractor algorithm, starting with properties of the `tangle_attractor_step` relation. The first property will be used later to show that the tangle attractor terminates.

Lemma 5.2. *X monotonically increases in the* `tangle_attractor_step` *relation.*

Proof. At each step of the relation, at least one node is added to X to obtain X'. We can show this with a structural induction on the cases of this relation. In the `player` case, this is a α-node that can move into X. In the `opponent` case, the added node is an $\overline{\alpha}$-node that must move into X. Finally, in the `tangle` case, we add all nodes in a tangle t, where the opponent can only escape into X. This t will always contain some nodes outside X because $t \setminus X \neq \emptyset$, so this case always adds at least one node. □

The Isabelle proof for this lemma, as well as for it holding for the reflexive transitive closure of the relation, are shown in Listing 5.12 below. Note that this monotonic increase takes the form of X being a strict subset of X' for the single step, but in the reflexive transitive closure, it is a non-strict subset of X'. This is because of the reflexivity in the reflexive transitive closure, where $X = X'$. This second proof also uses the induction rule `rtranclp_induct2` from the Isabelle/HOL standard library, which is specifically intended for relations on pairs.

```
lemma tangle_attractor_step_mono:
  "tangle_attractor_step (X,σ) (X',σ') ⟹ X ⊂ X'"
  by (induction rule: tangle_attractor_step_induct) auto

lemma tangle_attractor_step_rtranclp_mono:
  "tangle_attractor_step** (X,σ) (X',σ') ⟹ X ⊆ X'"
  apply (induction rule: rtranclp_induct2)
  using tangle_attractor_step_mono by blast+
```

Listing 5.12: Proving the tangle attractor step relation is monotonic on X.

The next lemma will allow us to show that our tangle attractor is always part of V. We will need this for various future properties. Note that we cannot say directly that the obtained X' is part of V. In this context, it is not known whether the original X was part of V. The Isabelle proofs of this property are shown in Listing 5.13.

Lemma 5.3. *The obtained X' in the* `tangle_attractor_step` *relation is always part of $X \cup V$.*

Proof. Each step of the relation adds nodes from V to X to obtain X'. We can once again show this with a structural induction on the cases of the step relation. In the `player` case, we add a α-node, which is part of V. In the `opponent` case, we add an $\overline{\alpha}$-node, which is also part of V. Finally, in the tangle case, we add a tangle t, which must also be part of V by the definition of a tangle. □

```
lemma tangle_attractor_step_ss:
  "tangle_attractor_step (X,σ) (X',σ') ⟹ X' ⊆ X ∪ V"
  apply (induction rule: tangle_attractor_step_induct)
  subgoal using V_α_subset by auto
  subgoal by simp
  subgoal using tangles_T player_tangle_in_V by auto
  done

lemma tangle_attractor_step_rtranclp_ss:
  "tangle_attractor_step** (X,σ) (X',σ') ⟹ X' ⊆ X ∪ V"
  apply (induction rule: rtranclp_induct2)
  using tangle_attractor_step_ss by blast+
```

Listing 5.13: The resulting X' of a tangle attractor step is part of $X \cup V$.

Our next lemma will be used to show that the attractor is finite. Eventually, this will be useful to show properties of `search`. The Isabelle proofs of this property are shown in Listing 5.14. Once again, the second proof uses the `rtranclp_induct2` induction rule.

Lemma 5.4. *If X is finite, our* `tangle_attractor_step` *relation gives an X' that is also finite.*

Proof. From Lemma 5.3, we know that X' is a subset of $X \cup V$. V is finite by our definition of a parity game, and from our assumption here, so is X. Therefore, their union is also finite. A subset of a finite set is finite, so X' is finite. □

```
lemma fin_tangle_attractor_step:
  "⟦tangle_attractor_step (X,σ) (X',σ'); finite X⟧ ⟹ finite X'"
  using finite_subset[OF tangle_attractor_step_ss] by blast

lemma fin_tangle_attractor_step_rtranclp:
  "⟦tangle_attractor_step** (X,σ) (X',σ'); finite X⟧ ⟹ finite X'"
  apply (induction rule: rtranclp_induct2)
  using fin_tangle_attractor_step by blast+
```

Listing 5.14: If the initial X is finite, the resulting X' is also finite.

Now, we begin with our lemmas proving correctness of `tangle_attractor_step` using its invariant. We start with the initial state at the start of the tangle attractor, where the invariant should hold. Listing 5.15 shows our Isabelle proof for this property. The majority of the proof is rather trivial and is immediately solved by `clarsimp`. For the sixth property, we need the fact that the origin of a lasso is part of that lasso. For the seventh, we need to make explicit that there always exists an empty path from a node to itself.

Lemma 5.5. *Our invariant* `tangle_attractor_step_I` *(Listing 5.11) holds for the base step, where X is target region A, and σ is an empty strategy.*

Proof. Our invariant requires six properties. The first of these is that A is a subset of X. Because $X = A$ in our base step, this is clearly true. The next property requires σ to be a valid strategy for α. An empty strategy is always valid for any player, so this is also true. The third property requires that the domain of σ is equal to $V_\alpha \cap (X \setminus A)$. Here, $X \setminus A = \emptyset$. The domain of an empty strategy is $\emptyset$, so this is once again true. The fourth property asks that the range of σ be a subset of X. An empty set is a subset of any set, so this is true. The fourth property requires that, under σ, the region X is partially closed in $X \setminus A$. $X \setminus A$ is an empty set, and this is a subset relation, so this is trivially true. Next, we need all plays starting in X to move to A. $X = A$, so every play starting in X is already in A, meaning this property is also satisfied. Finally, we need a path from all nodes in X to some node in A. Once again, $X = A$ means every node in X is in A, so an empty path satisfies this condition. □

```
lemma tangle_attractor_step_I_base:
  "tangle_attractor_step_I (A,Map.empty)"
  unfolding tangle_attractor_step_I_def split
  apply (clarsimp; intro conjI ballI)
  subgoal for x using origin_in_lasso[of _ x] by blast
  subgoal for x using path.simps(1)[of _ x x] by blast
  done
```

Listing 5.15: Our invariant holds for the base step.

The rest of our lemmas each prove one part of the invariant is preserved at each step. The first of these shows that our step gives a valid strategy.

Lemma 5.6. *If our invariant* `tangle_attractor_step_I` *(Listing 5.11) holds for the initial state, the strategy σ' obtained in our* `tangle_attractor_step` *relation is a valid strategy for α.*

Proof. We prove this property by structural induction on the cases of `tangle_attractor_step`. In the `player` case, we are adding a valid move from a *player*-node to a valid strategy for α. This means the result remains valid. In the `opponent` case, we do not change the strategy at all, so it remains valid. Finally, in the `tangle` case, we first restrict our valid tangle strategy, which means it remains valid. Then, we combine this with the existing valid strategy σ. Combining two valid strategies gives a valid strategy, so this case also preserves validity. □

The Isabelle proof for this lemma is shown in Listing 5.16. It follows much the same structure as the proof above, using automated provers.

```
lemma tangle_attractor_step_strat_of_Vα:
  "⟦tangle_attractor_step (X,σ) (X',σ'); tangle_attractor_step_I (X,σ)⟧
    ⟹ strategy_of Vα σ'"
  unfolding tangle_attractor_step_I_def split
  apply (induction rule: tangle_attractor_step_induct)
  subgoal by (simp add: strategy_of_overwrite)
  subgoal by fast
  subgoal by (simp add: strategy_of_restrict player_tangle_strat_def)
  done
```

Listing 5.16: The strategy obtained in our step is valid for α.

The following lemma concerns the domain of our witness strategy, which should lie within the attracted region. Listing 5.17 shows the Isabelle proof for this property. It only needs a structural induction on `attractor_step` and the definition of a tangle strategy to finish the proof using `auto`.

Lemma 5.7. *If our invariant* `tangle_attractor_step_I` *(Listing 5.11) holds for the initial state, the strategy σ' obtained in our* `tangle_attractor_step` *relation has a domain covering all α-nodes in $X' \setminus A$.*

Proof. The domain of our initial σ already covers all α-nodes in $X \setminus A$. We show that our step relation extends this strategy with all α-nodes in $X' \setminus X$ with a structural induction on its cases. In the `player` case, we have added a single α-node x to X to get X'. This node is assigned a valid successor y in X in our new strategy σ', so the domain of σ' once again covers all α-nodes in $X' \setminus A$. In the `opponent` case, the only node added to X is an $\overline{\alpha}$-node. This means the unaltered σ still covers all α-nodes in $X' \setminus A$. Finally, in the `tangle` case, we add all nodes in a tangle t to X to get X'. We limit τ to $t \setminus A$ and combine the result with σ to obtain our σ'. As a tangle strategy, τ covers all α-nodes in t. Therefore, τ limited to $t \setminus A$ covers all α-nodes in $t \setminus A$. When we combine this strategy with σ, we get a σ' that has in its domain all α-nodes in $X' \setminus A$. □

```
lemma tangle_attractor_step_dom:
  "⟦tangle_attractor_step (X,σ) (X',σ'); tangle_attractor_step_I (X,σ)⟧
    ⟹ dom σ' = Vα ∩ (X'-A)"
  unfolding tangle_attractor_step_I_def split
  apply (induction rule: tangle_attractor_step_induct)
  unfolding player_tangle_strat_def by auto
```

Listing 5.17: The domain of σ' obtained in our step is all α-nodes in the attractor not in A.

Our next lemma concerns the range of our witness strategy σ. This should lie within the attracted region.

Lemma 5.8. *If our invariant* `tangle_attractor_step_I` *(Listing 5.11) holds for the initial state, the strategy σ' obtained in our* `tangle_attractor_step` *relation has a domain range within X'.*

Proof. By the invariant, the initial σ already has a range within X. We show that the X' obtained through our step relation has a range within X' using a structural induction on its cases. In the `player` case, σ is extended with a single move from x to a y in X. As a result σ' has a range in X. Because $X \subset X'$, this means the range of σ' lies in X'. Next, in the `opponent` case, we do not change σ at all. The original σ has its range in X, and $X \subset X'$, so the range of σ' lies in X'. Finally, we have the `tangle` case. The tangle strategy τ has a range in t, which is part of X' because $X' = X \cup t$. This remains true when it is restricted to $t \setminus A$. It is then combined with σ, which has a range in X, which is also part of X'. This means the range of the combined σ' lies in X'. □

The Isabelle proof for this lemma is shown in Listing 5.18. It follows a structural induction like our proof above, but is able to make some of the deductions in the cases in a single step. Only the final case, the `tangle` case, needs an additional lemma about the range of restricted strategies, and the unfolding of `player_tangle_strat_def` and `ran_def` to complete the proof.

```
lemma tangle_attractor_step_ran:
  "⟦tangle_attractor_step (X,σ) (X',σ'); tangle_attractor_step_I (X,σ)⟧
    ⟹ ran σ' ⊆ X'"
  unfolding tangle_attractor_step_I_def split
  apply (induction rule: tangle_attractor_step_induct)
  subgoal by fastforce
  subgoal by blast
  subgoal for t X τ
    using ran_restrictD[of _ τ "t-A"]
    by (clarsimp simp: player_tangle_strat_def ran_def) blast
  done
```

Listing 5.18: The range of σ' obtained in our step lies within the attractor.

Next, we prove that the attracted region is partially closed, and remains so after each step of the algorithm. We prove this in two steps: first showing that the original region remains closed under the new strategy, then showing that the newly attracted region is partially closed.

Lemma 5.9. *If our invariant* `tangle_attractor_step_I` *(Listing 5.11) holds for the initial state, the region X is partially closed in $X \setminus A$ in the subgame $\mathcal{G}[\sigma']$ of the strategy σ' obtained in our* `tangle_attractor_step` *relation.*

Proof. We have a σ for which X is partially closed in $X \setminus A$ in $\mathcal{G}[\sigma]$ by our invariant. We show that the obtained σ' also makes X partially closed in $X \setminus A$ using a structural induction on the cases of `tangle_attractor_step`. In the `player` case, the added $[x \mapsto y]$ in our σ' does not affect the domain X. Consequently, all moves in the domain X are the same as they are in σ, so our partial closedness is retained. Next, in the `opponent` case, our strategy is unchanged, so the closedness is also unchanged. Finally, in the `tangle` case, the addition of σ to a restricted τ overwrites any moves in the domain X. Therefore, any moves in that domain are the same, and X remains partially closed in $X \setminus A$ under σ'. □

Our Isabelle proof is shown in Listing 5.19. It follows the same induction, needing an auxiliary lemma to show the `player` case. This lemma, `ind_subgraph_add_disjoint(1)` shows that, because the domains of σ and the added move $[x \mapsto y]$ are disjoint, $\mathcal{G}[\sigma']$ is a subgraph of $\mathcal{G}[\sigma]$. As a result, the closedness is unaffected. In the `tangle` case, it needs to unfold the definitions of `induced_subgraph` and `E_of_strat`, after which it can find out that $\mathcal{G}[\sigma']$ is a subgraph of $\mathcal{G}[\sigma]$, leading to the same deduction as before.

```
lemma tangle_attractor_step_closed_X:
  "⟦tangle_attractor_step (X,σ) (X',σ'); tangle_attractor_step_I (X,σ)⟧
    ⟹ induced_subgraph σ' (X-A) ⊆ X"
  unfolding tangle_attractor_step_I_def split
  apply (induction rule: tangle_attractor_step_induct)
  subgoal for x X y σ
    using ind_subgraph_add_disjoint(1)[of σ "[x↦y]"]
    by simp blast
  subgoal by blast
  subgoal
    unfolding induced_subgraph_def E_of_strat_def
    by simp blast
  done
```

Listing 5.19: Under σ' obtained in our step, the initial X is partially closed in $X \setminus A$.

Lemma 5.10. *If our invariant* `tangle_attractor_step_I` *(Listing 5.11) holds for the initial state, the region X' is partially closed in $X' \setminus A$ in the subgame $\mathcal{G}[\sigma']$ of the strategy σ' obtained in our* `tangle_attractor_step` *relation.*

Proof. We will follow along with our Isabelle proof for this lemma. This proof starts by setting the assumptions that we have a step from (X, σ) to (X, σ'), and that the invariant holds for (X, σ). It then follows from Lemma 5.7 that the domain of σ' is all α-nodes in X'. Per Lemma 5.8, the range of σ' lies in X'. Finally, from lemma 5.9 it follows that the region X is partially closed in $X \setminus A$ in $\mathcal{G}[\sigma']$.

```
lemma tangle_attractor_step_closed_X':
  "⟦tangle_attractor_step (X,σ) (X',σ'); tangle_attractor_step_I (X,σ)⟧
    ⟹ induced_subgraph σ' (X'-A) ⊆ X'"
proof -
  assume step: "tangle_attractor_step (X,σ) (X',σ')"
     and invar: "tangle_attractor_step_I (X,σ)"
  hence "dom σ' = V_α ∩ (X'-A)" "ran σ' ⊆ X'"
    "induced_subgraph σ' `` (X-A) ⊆ X"
    using tangle_attractor_step_dom tangle_attractor_step_ran
    tangle_attractor_closed_X
    by auto
```

We use these properties in a structural induction on the cases of `tangle_attractor_step`.

```
  from step invar this show ?thesis
    unfolding tangle_attractor_step_I_def split
  proof (induction rule: tangle_attractor_step_induct)
```

In the `player` case, we know that X was already partially closed in $X \setminus A$ in $\mathcal{G}[\sigma]'$. Because $X \subset X'$, this means X' is partially closed in $X \setminus A$. Our new move $[x \mapsto y]$ adds a successor in X' to the sole node in $X' \setminus X$. Together, this makes X' partially closed in $X' \setminus A$. Isabelle can deduce this if we unfold `induced_subgraph_def` and `E_of_strat_def`.

```
    case (player x X y σ) thus ?case
      unfolding induced_subgraph_def E_of_strat_def by auto
```

In the `opponent` case, the newly added x in X' only has successors in X. Because $X \subset X'$, this means it only has successors in X'. Combined with the fact that X was already partially closed in $X \setminus A$ in $\mathcal{G}[\sigma']$, this means that X' is partially closed in $X' \setminus A$ in $\mathcal{G}[\sigma']$. Isabelle once again deduces this immediately if we unfold the definitions of `induced_subgraph` and `E_of_strat`.

```
  next
    case (opponent x X σ) thus ?case
      unfolding induced_subgraph_def E_of_strat_def by blast
```

The `tangle` case is more complicated. Before we get to the proper proof, we use the `let` keyword to give shorthand for X', σ', and $\mathcal{G}[\sigma']$ to make our properties more readable. We also get the relevant

properties from our case, namely that all escapes of the tangle t are part of X, that the domain of σ' is all α-nodes in $X' \setminus A$, that the range of σ' lies in X', and that X is partially closed in $X \setminus A$ in $\mathcal{G}[\sigma']$.

```
  next
    case (tangle t X τ σ)
    let ?X' = "X ∪ t"
    let ?σ' = "τ |` (t-A) ++ σ"
    let ?Gσ' = induced_subgraph ?σ'"
    from tangle have
      escapes_in_X: "opponent_escapes t ⊆ X" and
      σ'_dom: "dom ?σ' = V_α ∩ (?X'-A)" and
      σ'_ran: "ran ?σ' ⊆ ?X'" and
      σ'_closed_X: "?Gσ' `` (X-A) ⊆ X"
      by blast+
```

Now, we can show that the property holds for this case. `clarsimp` gives us an edge (x, y) in the game, where x is either part of X or t, and it is not part of A. Furthermore, y is not part of t. We make a case distinction on whether x lies in X or in t. If x lies in x, we already know y is in X because X is partially closed in $X \setminus A$ under σ'. If x is part of t, we make another case distinction. If x is a α-node, then it is part of the domain of σ', so y must be in the range of σ', which is some subset of X'. If x is not a α-node, then y is one of the escapes of t. The escapes of t are all in X, so y is in X.

```
    show ?case
      apply clarsimp
      subgoal for y x
        apply (cases "x ∈ X")
        using σ'_closed_X apply blast
        apply (cases "x ∈ V_α")
        using σ'_dom σ'_ran ind_subgraph_edge_dst[of x y ?σ'] apply force
        using escapes_in_X opponent_escapes_pre[of y t] by force
      done
  qed
qed
```

With this, we have completed our proof that X' is partially closed in $X' \setminus A$ in $\mathcal{G}[\sigma']$. □

We now prove that the obtained strategy makes it so that all plays in its subgame that start in the attracted region are either led into A, or are won by α. This is once again proven in two parts, one showing this property for the initial region, and one showing it for the new attracted region.

Lemma 5.11. *If our invariant* `tangle_attractor_step_I` *(Listing 5.11) holds for the initial state, we obtain a strategy σ' in our* `tangle_attractor_step` *relation such that all plays starting in the region X either move to A, or are won by α in $\mathcal{G}[\sigma']$.*

Proof. We will again follow our Isabelle proof for this lemma. This proof starts an induction over the cases of our `tangle_attractor_step` relation.

```
lemma tangle_attractor_step_forces_A_or_wins_X:
  "⟦tangle_attractor_step (X,σ) (X',σ'); tangle_attractor_step_I (X,σ)⟧
    ⟹ (∀x∈X. ∀xs ys. lasoo (induced_subgraph σ') x xs ys
      ⟶ (set (xs@ys) ∩ A ≠ {} ∨ winning_player ys))"
  unfolding tangle_attractor_step_I_def split
proof (induction rule: tangle_attractor_step_induct)
```

In the `player` case, we know that any play starting from X is the same in $\mathcal{G}[\sigma']$ as it was in $\mathcal{G}[\sigma]$ because the move $[x \mapsto y]$ does not affect any of the moves in X. Therefore, all steps from there should be the same, thus our property that held for σ by the invariant, should also hold for σ'. In Isabelle, we show that $[x \mapsto y]$ does not overlap with the domain of σ. This means a lasso in $\mathcal{G}[\sigma']$ must also exist in $\mathcal{G}[\sigma]$. The case is then proven using `metis` with some additional lemmas to translate between `σ(x↦y)` and `σ ++ [x↦y]`.

```
  case (player x X y σ)
  hence disj: "dom σ ∩ dom [x↦y] = {}" by simp
  from player show ?case
    using subgraph_lasso[OF ind_subgraph_add_disjoint(1)[OF disj]]
    by (metis map_add_empty map_add_upd)
```

In the `opponent` case, the two strategies are identical, so this property is entirely unaffected. Isabelle solves this immediately.

```
next
  case (opponent x X σ) thus ?case by blast
```

Finally, in the `tangle` case, $\mathcal{G}[\sigma']$ is a subgame of $\mathcal{G}[\sigma]$. This is because we overwrite all moves in the tangle strategy τ that overlap with moves in σ with those from σ. As a result, any lasso that exists in $\mathcal{G}[\sigma']$ also exists in $\mathcal{G}[\sigma]$ and exhibits the same properties. In our Isabelle proof, we define some shorthand for readability. Then, we show that $\mathcal{G}[\sigma']$ is a subgame of $\mathcal{G}[\sigma]$. Finally, we prove the property for this case using a lemma that shows that every lasso in $\mathcal{G}[\sigma']$ also exists in $\mathcal{G}[\sigma]$.

```
next
  case (tangle t X τ σ)
  let ?σ' = "τ |` (t-A) ++ σ"
  have subgraph: "induced_subgraph ?σ' ⊆ induced_subgraph σ"
    unfolding induced_subgraph_def E_of_strat_def by auto
  from tangle show ?case
    using subgraph_lasso[OF subgraph] by fast
qed
```

This completes our proof that all plays starting in X in $\mathcal{G}[\sigma']$ either move to A, or are won by α. ☐

Lemma 5.12. *If our invariant* `tangle_attractor_step_I` *(Listing 5.11) holds for the initial state, we obtain a strategy σ' in our* `tangle_attractor_step` *relation such that all plays starting in the region X' either move to A, or are won by α in $\mathcal{G}[\sigma']$.*

Proof. We once again follow our Isabelle proof for this lemma. This proof starts with the assumptions that we have a step from (X, σ) to (X', σ'), and that the invariant holds for (X, σ). By Lemma 5.7, the domain of σ' is all α-nodes in X'. Furthermore, by Lemmas 5.9 and 5.10, X is partially closed in $X \setminus A$ in $\mathcal{G}[\sigma']$, and X' is partially closed in $X' \setminus A$ in $\mathcal{G}[\sigma']$. Finally, every play in $\mathcal{G}[\sigma']$ that starts in X either moves to A or is won by α.

```
lemma tangle_attractor_step_forces_A_or_wins:
  "⟦tangle_attractor_step (X,σ) (X',σ'); tangle_attractor_step_I (X,σ)⟧
    ⟹ (∀x∈X'. ∀xs ys. lasoo (induced_subgraph σ') x xs ys
      ⟶ (set (xs@ys) ∩ A ≠ {} ∨ winning_player ys))"
proof -
  assume step: "tangle_attractor_step (X,σ) (X',σ')" and
    invar: "tangle_attractor_step_I (X,σ)"

  hence props:
    "dom σ' = V_α ∩ (X'-A)"
    "induced_subgraph σ' `` (X-A) ⊆ X"
    "induced_subgraph σ' `` (X'-A) ⊆ X'"
    "∀x∈X. ∀xs ys. lasso (induced_subgraph σ' x xs ys
      ⟶ set (xs@ys) ∩ A ≠ {} ∨ winning_player ys"
    using tangle_attractor_step_dom
       tangle_attractor_step_closed_X
       tangle_attractor_step_closed_X'
       tangle_attractor_step_forces_A_or_wins_X
    by auto
```

Whenever we add a single node to X to get X', and that node only has successors in X, then any lasso starting in X' either entirely exists in X, or starts on x. If it starts on x, it takes a single step to

get to X, and the remainder of the lasso is contained entirely within X, so that property also holds from there.

We define an auxiliary lemma within our proof so we can reuse this reasoning for both the `player` and `opponent` cases. This lemma starts with the fixing of X, the inserted node x, the strategy σ, and a lasso starting in a v with spoke xs and loop ys. We also define shorthand for X', which is X with x inserted into it, and the subgame $\mathcal{G}[\sigma]$. Then we give our assumptions, namely that x only has successors in X; x is not part of X; X is partially closed in $X \setminus A$ in $\mathcal{G}[\sigma]$; X' is partially closed in $X' \setminus A$ in $\mathcal{G}[\sigma]$; that all plays starting in X in $\mathcal{G}[\sigma]$ either move to A, or are won by α; that v is a node in X'; and that we have our lasso from v with spoke xs and loop ys in $\mathcal{G}[\sigma]$. We also make the assumption that the lasso does not go to A, meaning we will only have to prove that this lasso is won by α.

```
{
  fix X :: "'v set" and x :: 'v
  fix σ v xs ys
  let ?X' = "insert x X"
  let ?Gσ = "induced_subgraph σ"
  assume x_succs_in_X: "∀y. (x,y) ∈ ?Gσ ⟶ y ∈ X"
     and x_notin_X: "x ∉ X"
     and σ_closed_X: "?Gσ `` (X-A) ⊆ X"
     and σ_closed_X': "?Gσ `` (?X'-A) ⊆ ?X'"
     and σ_forces_A_or_wins_X:
     "∀x∈X. ∀ xs ys. lasso ?Gσ x xs ys
        ⟶ set (xs@ys) ∩ A ≠ {} ∨ winning_player ys"
     and v_in_X': "v ∈ ?X'"
     and lasso: "lasso ?Gσ v xs ys"
     and lasso_no_A: "set (xs@ys) ∩ A = {}"
```

We separate the assumption that the lasso does not intersect with A into the facts that the spoke and loop do not intersect with A independently. Because v is part of X' and included in the lasso, it must now be part of $X' \setminus A$.

```
  hence xs_no_A: "set xs ∩ A = {}"
    and ys_no_A: "set ys ∩ A - {}" by auto
  with <v∈?X'> have "v∈? X'-A"
    using origin_in_lasso[OF lasso] by blast
```

Now, we obtain a v', which is the start of our loop ys. We show that this loop does not contain x, because it would have to then move to a successor of x, which is part of X. Because X is partially closed in $X \setminus A$ in $\mathcal{G}[\sigma']$, and we have our assumption that there is no A in the loop, the whole loop is contained in the region $X \setminus A$. As a consequence x cannot be part of the loop, because x is not part of X.

```
  from lasso obtain v' where cycle: "cycle ?Gσ v' ys"
    unfolding lasso_def by auto

  have ys_no_X: "x ∉ set ys" proof
    assume "x ∈ set ys"

    from x_succs_X have X_in_ys: "set ys ∩ X ≠ {}"
      using cycle_intermediate_node[of cycle <x∈set ys>] cycle_D[of ?Gσ]
      by (metis disjoint_iff[of "set ys" X]

    with cycle ys_no_A σ_closed_X
    have "set ys ⊆ X"
      using cycle_intersects_partially_closed_region[of ?Gσ]
      by blast

    with <x∈set ys> <x∉X> show False by blast
  qed
```

Moreover, the entire loop ys is contained within $X' \setminus A$. This is because of the partial closedness of X' in $X' \setminus A$ in $\mathcal{G}[\sigma']$. The lasso starts with v, which is part of $X' \setminus A$, and from there it will stay in $X' \setminus A$ at every step, as neither the spoke or the loop contain A.

```
    moreover from <v∈?X'-A> σ_closed_X' xs_no_A ys_no_A lasso
    have "set ys ⊆ ?X'-A"
      using lasso_partially_closed_sets[of v] by simp
```

Because ys is contained within $X' \setminus A$, and does not contain x, it is entirely contained within X. As a consequence, v' is also part of X.

```
    ultimately have "set ys ⊆ X" by blast
    hence "v' ∈ X" using origin_in_cycle[OF cycle] by blast
```

Now we have a cycle that starts in X. A cycle is a lasso without a loop, so we can translate the cycle back to a lasso. This lasso starts in X and does not contain A, so by the property that all lassos starting in X either move to A are won by α, this lasso is won by α. With this, we have completed the auxiliary lemma, that we name `add_single_node`.

```
    with σ_forces_A_or_wins_X cycle ys_no_A have "winning_player ys"
      using cycle_iff_lasso[of ?Gσ v' ys] by fastforce
  } note add_single_node=this
```

With this lemma defined, we start our induction using the properties we showed earlier. In the `player` case, we can show that all successors of x in $\mathcal{G}[\sigma']$ are part of X. The only successor is y, which is part of X. With the closedness properties and the property that all plays starting in X are forced to move to A or be won by α, we can now show that our thesis holds in this case using our lemma `add_single_node`.

```
  from step invar props show ?thesis
    unfolding tangle_attractor_step_I_def split
  proof (induction rule: tangle_attractor_step_induct)
    case (player x X y σ)
    let ?X' = "insert x X"
    let ?Gσ' = "induced_subgraph (σ(x↦y))"

    from player have
      "?Gσ' `` (X-A) ⊆ X"
      "?Gσ' `` (?X'-A) ⊆ ?X'"
      "∀x∈X. ∀xs ys. lasso ?Gσ' x xs ys
        ⟶ set (xs@ys) ∩ A ≠ {} ∨ winning_player ys"
      by blast+

    moreover from <y∈X> have "∀y. (x,y) ∈ ?Gσ' ⟶ y ∈ X"
      using ind_subgraph_to_strategy by fastforce

    ultimately show ?case using add_single_node by blast
  next
```

Our proof for the `opponent` case is much the same. We first get our properties of closedness, and the fact that all plays starting in X are forced under σ' to move to A or be won by α. Then, we show that all successors of x in $\mathcal{G}[\sigma']$ are part of X. With these properties, we use our auxiliary lemma to show that the thesis holds for the case.

```
    case (opponent x X σ)
    let ?X' = "insert x X"
    let ?Gσ = "induced_subgraph σ"

    from opponent have
      "?Gσ `` (X-A) ⊆ X"
      "?Gσ `` (?X'-A) ⊆ ?X'"
      "∀x∈X. ∀xs ys. lasso ?Gσ x xs ys
```

```
      ⟶ set (xs@ys) ∩ A ≠ {} ∨ winning_player ys"
    by blast+

  moreover from <∀y. (x,y)∈E ⟶ y∈X> have
    "∀y. (x,y) ∈ ?Gσ ⟶ y ∈ X" by simp

  ultimately show ?case using add_single_node by blast
next
```

The `tangle` case cannot use our auxiliary lemma, so it requires more steps. We start by taking the relevant properties. These are mostly the same as the ones used in the prior two cases, but we also use properties of tangles, such as tangles being part of V, and all cycles in a tangle being won by α under the strategy τ.

```
  case (tangle t X τ σ)
  let ?X' = "X ∪ t"
  let ?σ' = "τ |` (t-A) ++ σ"
  let ?Gσ' = "induced_subgraph ?σ"
  let ?Gt = "player_tangle_subgraph t τ"

  from tangle have
    A_in_X: "A ⊆ X" and
    σ_dom: "dom σ = V_α ∩ (X-A)" and
    σ'_dom: "dom σ' = V_α ∩ (?X'-A)" and
    σ'_closed_X: "?Gσ' `` (X-A) ⊆ X" and
    σ'_closed_X': "?Gσ' `` (?X'-A) ⊆ ?X'" and
    σ'_forces_A_or_wins:
      "∀x∈X. ∀xs ys. lasso ?Gσ' x xs ys
        ⟶ set (xs@ys) ∩ A ≠ {} ∨ winning_player ys"
    by blast+

  from tangle have "t ⊆ V"
    using player_tangle_in_V tangles_T by auto
  from tangle have τ_winning:
    "∀v∈t. ∀xs. cycle ?Gt v xs ⟶ winning_player xs"
    unfolding player_tangle_strat_def Let_def by auto
```

We complete our proof by once again taking a lasso that does not intersect with A. We will consider two cases: one where the loop ys does not intersect with X, and one where it does. In the former case, since X is partially closed in $X \setminus A$ under σ', it is wholly contained in A. This makes it a lasso without a spoke that starts in X, and does not move to A. Because plays starting in X that do not intersect with A are won by α, this play is also won by α.

```
  {
    fix v xs ys
    assume v_in_X': "v ∈ ?X'"
       and lasso: "lasso ?Gσ' v xs ys"
       and lasso_no_A: "set (xs@ys) ∩ A = {}"
    hence xs_no_A: "set xs ∩ A = {}"
      and ys_no_A: "set ys ∩ A = {}" by auto
    with <v∈?X'> have "v ∈ ?X'-A"
      using origin_in_lasso[OF lasso] by blast
    from lasso obtain v' where cycle: "cycle ?Gσ' v' ys"
      unfolding lasso_def by auto

    consider (ys_has_X) "set ys ∩ X ≠ {}" | (ys_no_X) "set ys ∩ X = {}" by blast
    hence "winning_player ys" proof cases
      case ys_has_X
      with cycle ys_no_A σ'_closed_X have "set ys ⊆ X"
```

```
        using cycle_intersects_partially_closed_region[of ?Gσ']
        by blast
      hence "v' ∈ X" using origin_in_cycle[OF cycle] by blast

      with σ'_forces_A_or_wins cycle ys_no_A show ?thesis
        using cycle_iff_lasso[of ?Gσ' v' ys] by fastforce
    next
```

If our cycle ys does not contain some part of X, then the whole cycle is contained in the tangle t. All cycles in this tangle are won by α, including our ys. We will need to show that the cycle also exists in the tangle subgraph of t and τ, after which we can complete the proof.

```
      case ys_no_X
      from <v∈X'-A> σ'_closed_X' xs_no_A ys_no_A lasso
      have "set ys ⊆ ?X'-A"
        using lasso_partially_closed_sets[of v] by simp
      with ys_no_X have "set ys ⊆ t-X" by blast
      hence "v' ∈ t" using origin_in_cycle[OF cycle] by blast

      from σ'_dom σ_dom <t⊆V> <A⊆V>
      have subset:
        "Restr ?Gσ' (t-X) ⊆ ?Gt"
        unfolding induced_subgraph_def
             player_tangle_subgraph_def
             E_of_strat_def
        by auto
      from subgraph_cycle[OF subset cycle_restr_V[OF cycle <set ys⊆t-X>]]
      have "cycle ?Gt v' ys" by fast

      with τ_winning <v'∈t> show ?thesis by blast
    qed
  }
  thus ?case by blast
 qed
qed
```

With this, we have shown that all plays starting in X' in $\mathcal{G}[\sigma']$ either move to A, or are won by α. □

To show that there always exists a path from any node in X' to a node in A in $\mathcal{G}[\sigma']$, we once again first prove that this is true for any node in X.

Lemma 5.13. *If our invariant* `tangle_attractor_step_I` *(Listing 5.11) holds for the initial state, we obtain a strategy σ' in our* `tangle_attractor_step` *relation such that there exists a path from any node in X that leads to some node in A in $\mathcal{G}[\sigma']$.*

Proof. Once again, we follow our Isabelle proof. We first set our assumption that we have a step from (X, σ) to (X', σ'), and our invariant holds for (X, σ). From Lemma 5.7, we get the fact that the domain of σ' is all α-nodes in $X' \setminus A$.

```
lemma tangle_attractor_step_path_X_to_A:
  "⟦tangle_attractor_step (X,σ) (X',σ'); tangle_attractor_step_I (X,σ)⟧
    ⟹ (∀x∈X. ∃x'∈A. ∃xs. path (induced_subgraph σ') x xs x')"
proof -
  assume step: "tangle_attractor_step (X,σ) (X',σ')"
     and invar: "tangle_attractor_step_I (X,σ)"
  hence dom: "dom σ' = V_α ∩ (X'-A)"
    using tangle_attractor_step_dom by auto
```

We define an auxiliary lemma showing that if we have two strategies σ and σ', where a region X is partially closed in $X \setminus A$ in $\mathcal{G}[\sigma]$, $\mathcal{G}[\sigma]$ is a subgraph of $\mathcal{G}[\sigma']$ within X, and there exists a path from any node in X to some node in A in $\mathcal{G}[\sigma]$, then there also exists a path from any node in X to some

node in A in $\mathcal{G}[\sigma']$. We will be using this lemma in our upcoming induction, where it will help solve the `player` and `tangle` cases.

The lemma starts with fixing our X, σ, and σ'. We then use the `let` keyword to define shorthand for $\mathcal{G}[\sigma]$ and $\mathcal{G}[\sigma']$, after which we set our aforementioned assumptions.

```
{
  fix X :: "'v set" and σ σ' :: "'v strat"
  let ?Gσ = "induced_subgraph σ"
  let ?Gσ' = "induced_subgraph σ'"
  assume σ_closed: "?Gσ `` (X-A) ⊆ X"
     and restr_subgraph: "Restr ?Gσ X ⊆ ?Gσ'"
     and path_ex: "∀x∈X. ∃y∈A. ∃xs. path ?Gσ x xs y"
```

Next, we start our proof for the existence of a path from any node in X to some node in A in $\mathcal{G}[\sigma']$. In this proof, we fix our x as a node in X, and consider two cases: either x is a node in A, or it is not a node in A.

```
  have "∀x∈X. ∃y∈A. ∃xs. path Gσ' x xs y"
  proof (rule ballI)
    fix x assume x_in_X: "x ∈ X"
    consider (x_in_A) "x ∈ A" | (x_notin_A) "x ∉ A" by blast
    thus "∃y∈A. ∃xs. path ?Gσ' x xs y" proof cases
```

In the case where x is a node in A, we already have a path from x to some node in A. This is the empty path from x to itself.

```
      case x_in_A
      show ?thesis
        apply (rule bexI[where x=x])
        apply (rule exI[where x="[]"])
        using x_in_A by auto
```

In the case where x is not a node in A, we have an x in $X \setminus A$. We can obtain a path in $\mathcal{G}[\sigma]$ from x to some node in A from our assumptions. We use a lemma, `shortest_subpath_to_region` to get the shortest part of this path that leads to A without intersecting with A, which is the path xs to y, where $y \in A$.

```
    next
      case x_notin_A
      with <x∈X> have "x∈x-A" by simp
      from <x∈X> path_ex obtain xs y where
        y_in_A: "y ∈ A" and
        xs_no_A: "set xs ∩ A = {}" and
        path: "path ?Gσ x xs y"
        using shortest_subpath_to_region[of ?Gσ] by metis
```

Because this path does not intersect with A, x is not part of A, and our X is partially closed in $X \setminus A$ in $\mathcal{G}[\sigma]$, this whole path is contained within the region X.

```
      from path_partially_closed[OF <x∈X-A> σ_closed path xs_no_A]
      have xs_in_X: "set xs ⊆ X" and "y ∈ X" by blast+
```

One of our assumptions states that $\mathcal{G}[\sigma]$ is a subgraph of $\mathcal{G}[\sigma']$ within the region X. Our path is entirely contained within X, so it also exists in $\mathcal{G}[\sigma']$. This means xs forms a path from X to A in $\mathcal{G}[\sigma']$, completing the proof of our lemma. We name this lemma `X_path_to_A`.

```
      from path_restr_V[OF path xs_in_X <y∈X>]
      have "path ?Gσ' x xs y"
        using subgraph_path[OF restr_subgraph] by simp

      with <y∈A> show ?thesis by blast
    qed
  qed
} note X_path_to_A = this
```

With this lemma defined, we use a structural induction on the cases of our step relation. This induction also uses the domain property we obtained before we proved our lemma.

```
from step invar dom show ?thesis
  unfolding tangle_attractor_step_I_def split
proof (induction rule: tangle_attractor_step_induct)
```

In the `player` case, we first define some shorthand. Now, the case itself tells us that X is partially closed in $X \setminus A$ in $\mathcal{G}[\sigma]$, and that we can obtain a path in $\mathcal{G}[\sigma]$ from any node in X to some node in A.

```
  case (player x X y σ)
  let ?X' = "insert x X"
  let ?σ' = "σ(x↦y)"
  let ?Gσ = "induced_subgraph σ"
  let ?Gσ' = "induced_subgraph σ'"

  from player have
    σ_closed: "?Gσ' `` (X-A) ⊆ X" and
    path_ex: "∀x∈X. ∃x'∈A. ∃xs. path ?Gσ x xs x'"
    by blast+
```

Because our added move $[x \mapsto y]$ does not affect the moves within X, the graph $\mathcal{G}[\sigma']$ is identical to $\mathcal{G}[\sigma]$ within the region X. This means that $\mathcal{G}[\sigma]$ is a subgraph of $\mathcal{G}[\sigma']$ within X.

```
  from <x∈Vα-X> have "Restr ?Gσ X ⊆ ?Gσ'"
    unfolding induced_subgraph_def E_of_strat_def
    by (auto split: if_splits)
```

With this, we have obtained all the properties we needed for our auxiliary lemma `X_path_to_A`. We can obtain a path in $\mathcal{G}[\sigma]$ from any node in X to A, and because these paths either start in A or stay contained in the region X, any such path also exists in $\mathcal{G}[\sigma']$. This completes our proof for the `player` case.

```
  from X_path_to_A[OF σ_closed this path_ex] show ?case by blast
```

In the `opponent` case, the strategy σ remains unchanged, so our thesis trivially holds.

```
next
  case (opponent x X σ) thus ?case by blast
```

In the `tangle` case, we start our proof with shorthand again. Then, we get from our case's assumptions that the domain of σ is all α-nodes in $X \setminus A$, that the domain of σ' is all α-nodes in $X' \setminus A$, that the region X is partially closed in $X \setminus A$ in $\mathcal{G}[\sigma]$, and that we can obtain a path in $\mathcal{G}[\sigma]$ from any node in X to some node in A.

```
next
  case (tangle t X τ σ)
  let ?X' = "X∪t"
  let ?σ' = "τ |` (t-A) ++ σ"
  let ?Gσ = "induced_subgraph σ"
  let ?Gσ' = "induced_subgraph σ'"

  from tangle have
    σ_dom: "dom σ = Vα ∩ (X-A)" and
    σ'_dom: "dom σ' = Vα ∩ (?X'-A)" and
    σ_closed: "?Gσ `` (X-A) ⊆ X" and
    path_ex: "∀x∈X. ∃y∈A. ∃xs. path ?Gσ x xs y"
    by blast+
```

$\mathcal{G}[\sigma]$ is a subgraph of $\mathcal{G}[\sigma']$ within X because we have overwritten all of τ in X with moves from the original σ. Isabelle needs the domains to show this, along with the unfolding of the definitions of `induced_subgraph` and `E_of_strat`.

```
  from σ_dom σ'_dom have "Restr Gσ X ⊆ Gσ'"
    unfolding induced_subgraph_def E_of_strat_def by auto
```

Now, the thesis is true for this case by our auxiliary lemma once again.

```
    from X_path_to_A[OF σ_closed this path_ex] show ?case by blast
  qed
qed
```

This completes our proof, showing that we can obtain a path to A from any node in X in $\mathcal{G}[\sigma']$. □

We will now use this lemma to prove that we also have a path from any node in X' to a node in A in $\mathcal{G}[\sigma']$.

Lemma 5.14. *If our invariant* `tangle_attractor_step_I` *(Listing 5.11) holds for the initial state, we obtain a strategy σ' in our* `tangle_attractor_step` *relation such that there exists a path from any node in X' that leads to some node in A in $\mathcal{G}[\sigma']$.*

Proof. We will follow our Isabelle proof for this lemma. This proof starts by setting the assumptions that we have a step from (X, σ) to (X', σ'), and that the invariant holds for (X, σ). By Lemma 5.7 we know that the domain of σ' is all α-nodes in $X' \setminus A$. Furthermore, by Lemma 5.9 we know that X is partially closed in $X \setminus A$ in $\mathcal{G}[\sigma']$, and from Lemma 5.10 we know that X' is partially closed in $X' \setminus A$ in $\mathcal{G}[\sigma']$. Finally, from Lemma 5.13, we know that we can obtain a path in $\mathcal{G}[\sigma']$ from any node in X to some node in A. We obtain these properties in our Isabelle proof.

```
lemma tangle_attractor_step_path_to_A;
  "⟦tangle_attractor_step (X,σ) (X',σ'); tangle_attractor_step_I (X,σ)⟧
    ⟹ ∀x∈X'. ∃x'∈A. ∃xs. path (induced_subgraph σ') x xs x'"
proof -
  assume step: "tangle_attractor_step (X,σ) (X',σ')"
     and invar: "tangle_attractor_step_I (X,σ)"
  hence props:
    "dom σ' = V_α ∩ (X'-A)"
    "induced_subgraph (σ' `` (X-A) ⊆ X"
    "induced_subgraph (σ' `` (X'-A) ⊆ X'"
    "∀x∈X. ∃y∈A. ∃xs. path (induced_subgraph σ') x xs y"
    using tangle_attractor_step_dom
       tangle_attractor_step_closed_X
       tangle_attractor_step_closed_X'
       tangle_attractor_step_path_X_to_A
    by auto
```

We define an auxiliary lemma to make our upcoming induction proofs shorter. We have a node x, a strategy σ, and a path xs in $\mathcal{G}[\sigma]$ to a y in X. If we can obtain a path ys in $\mathcal{G}[\sigma]$ from any node in X to a node z in A, then we can append ys to xs to get a path in $\mathcal{G}[\sigma]$ from x to z. This means there always exists a path in $\mathcal{G}[\sigma]$ from x to some node in A. We name this lemma `append_path_to_A`.

```
  {
    fix X :: "'v set" and x y :: 'v and σ xs
    let ?Gσ = "induced_subgraph σ"
    assume y_in_X: "y∈X"
       and path_y: "path ?Gσ x xs y"
       and path_ex: "∀x∈X. ∃y∈A. ∃xs. path ?Gσ x xs y"
    then obtain z ys where
      "z∈A" "path ?Gσ x (xs@ys) z" by blast
    with path_y have "path ?Gσ x (xs@ys) z" by auto
    with <z∈A> have "∃y∈A. ∃xs. path ?Gσ x xs y" by blast
  } note append_path_to_A=this
```

Now, we start our induction with the previously obtained properties.

```
  from step invar props show ?thesis
    unfolding tangle_attractor_step_I_def split
  proof (induction rule: tangle_attractor_step_induct)
```

In the `player` case, we start by defining some shorthand. We then start our proof with some x' in X'. This x' may either be a node in X, or it is the x from our case.

```
case (player x X y σ)
let ?X' = "insert x X"
let ?σ' = "σ(x↦y)"
let ?Gσ' = "induced_subgraph σ'"
from player have X_path_to_A: "∀x∈X. ∃y∈A. ∃xs. path ?Gσ' x xs y" by blast

show ?case proof (rule ballI)
  fix x' assume x'_in_X: "x' ∈ ?X'"
  then consider (x'_in_X) "x' ∈ X" | (x'_is_x) "x' = x" by blast
  thus "∃y'∈A. ∃xs. path ?Gσ' x' xs y'" proof cases
```

In the case that x' is a node in X, we already know we have a path to A.

```
    case x'_in_X with X_path_to_A show ?thesis by blast
```

In the case that x' is x, we know x has a path to y in one step from itself. Because y is in X, we can use our auxiliary lemma to combine the path from x to y and the path from y to some node in A. This finishes our proof for the `player` case.

```
  next
    case x'_is_x
    with player have "path ?Gσ' x [x] y"
      using strategy_to_ind_subgraph[of ?σ' x y] by simp
    with X_path_to_A x'_is_x <y∈X> show ?thesis
    using append_path_to_A by blast
  qed
qed
```

In the `opponent` case, we follow a similar proof. We start with some shorthand, before we distinguish between the case where we have an x' in X, or an x' that is x.

```
next
  case (opponent x X σ)
  let ?X' = "insert x X"
  let ?Gσ = "induced_subgraph σ"
  from opponent have
    σ_dom: "dom σ = V_α ∩ (X-A)" and
    X_path_to_A: "∀x∈X. ∃y∈A. ∃xs. path ?Gσ x xs y"

  show ?case proof (rule ballI)
    fix x' assume x'_in_X': "x' ∈ ?X'"
    then consider (x'_in_X) "x' ∈ X" | (x'_is_x) "x' = x" by blast
    thus "∃y'∈A. ∃xs. path ?Gσ x' xs y" proof cases
```

In the former case, we once again know that there exists a path to A, making this case rather simple.

```
    case x'_in_X with X_path_to_A show ?thesis by blast
```

In the latter case, there once again exists a path from x to y in one step, and these paths can be combined. This completes our proof for the `opponent` case.

```
  next
    case x'_is_x
    with <x∈V_ᾱ> σ_dom have "path ?Gσ x [x] y"
      using ind_subgraph_notin_dom[of x y σ] by force
    with X_path_to_A x'_is_x <y∈X> show ?thesis
      using append_path_to_A by blast
  qed
qed
```

Finally, we have the `tangle` case. This case is more complicated than the others, but follows a similar structure. We start by defining shorthand and distinguishing between the cases where x is a node in X or a node in $t \setminus X$.

```
next
  case (tangle t X τ σ)
  let ?X' = "X∪t"
  let ?σ' = "τ |` (t-A) ++ σ"
  let ?Gτ = "induced_subgraph τ"
  let ?Gσ' = "induced_subgraph ?σ'"
  from tangle have
    σ_dom: "dom σ = Vα ∩(X − A)" and
    σ'_dom: "dom ?σ' = Vα ∩ (X ∪ t-A)" and
    X_path_to_A: "∀x∈X. ∃y∈A. ∃xs. path ?Gσ' x xs y"
    by blast+

  show ?case proof (rule ballI)
    fix x assume x_in_X': "x ∈ ?X'"
    then consider (x_in_X) "x ∈ X" | (x_in_t_min_X) "x ∈ t-X" by blast
    thus "∃x'∈A. ∃xs. path ?Gσ' x xs x'" proof cases
```

The case where x is a node in X is trivial as before, since we already know there exists a path from this node to A.

```
      case x_in_X with X_path_to_A show ?thesis by blast
```

The case where x is a node in the tangle t, but not in X, is much more complex. The proof operates on the general idea that a tangle is strongly connected, and that all escapes from t, of which there is at least one, lie in X. This means we can always get a path to a y in t from which the opponent can escape to a z in X. From z, we have a path to a node in A. This means we can combine the path to y with the single step from y to z, and the path from z to a node in A. However, there are a few complications caused by the possibility that X intersects with t, as well as the possibility that x is already a node from which the opponent can escape to X.

We start our proof for the case with some facts we will need later in the proof. t is a tangle, so it is part of V, the domain of its strategy τ is all of V_α in t, its range lies in t, and $\mathcal{G}[\tau]$ is strongly connected within t. Another fact that will become important later, is that $\mathcal{G}[\tau]$ restricted to $t \setminus X$ is a subgraph of $\mathcal{G}[\sigma']$.

```
    next
      case x_in_t_min_X
      from <t∈T> tangles_T have "t ⊆ V"
        using player_tangle_in_V by blast
      with <player_tangle_strat t τ> have
        τ_dom: "dom τ = t ∩ Vα" and
        τ_ran: "ran τ ⊆ t" and
        t_connected: "strongly_connected (Restr ?Gτ t) t"
        unfolding player_tangle_strat_def Let_def
        by (auto simp: player_tangle_subgraph_is_restricted_ind_subgraph)

      from σ_dom have subgraph_τ: "Restr ?Gτ (t-X) ⊆ ?Gσ'"
        unfolding induced_subgraph_def E_of_strat_def
        by (auto simp: map_add_dom_app_simps(3))
```

We define another auxiliary lemma to make further steps simpler. In this lemma, we assume we have a z in X and a nonempty path in $\mathcal{G}[\tau]$ restricted to t from x to z. We also assume this path does not intersect with X. We will show that this path also exists in $\mathcal{G}[\sigma']$.

```
{
  fix xs z
  assume z_in_X: "z ∈ X"
     and xs_notempty: "xs ≠ []"
     and xs_no_X: "set xs ∩ X = {}"
     and path: "path (Restr ?Gτ t) x xs z"
```

First, we obtain the subpath to the node y that leads to z. This y is not part of X, since the path xs does not intersect with X. Furthermore, this path and the edge between y and z exist within $\mathcal{G}[\tau]$.

```
  then obtain xs' y where
    xs_comp: "xs = xs' @ [y]" and
    xs'_no_X: "set xs' ∩ X = {}" and
    y_notin_X: "y ∉ X" and
    y_in_xs: "y ∈ set xs" and
    edge: "(y,z) ∈ ?Gτ" and
    path_y: "path ?Gτ x xs' y"
    using path_D_rev[OF path xs_notempty] path_inter(1)
    by force
```

We show that the path from x to y exists in $\mathcal{G}[\sigma']$. First, because the original path is restricted to t, the subpath xs' does not intersect with X, and y is not in X, xs and y both lie in $t \setminus X$. Consequently, since we know that $\mathcal{G}[\tau]$ restricted to $t \setminus X$ is a subgraph of $\mathcal{G}[\sigma']$, this path exists in $\mathcal{G}[\sigma']$.

```
  from restr_V_path[OF path xs_notempty]
    y_in_xs <y∉X> xs_comp xs'_no_X
  have "set xs' ⊆ t-X" "y ∈ t-X"
    by auto

  from path_restr_V[OF path_y this]
  have path_1: "path ?Gσ' x xs' y"
    using subgraph_path[OF subgraph_τ] by blast
```

Next, because the edge from y to z lies outside the domain of both σ and τ, and y is not part of X, y is a single-step path to z in $\mathcal{G}[\sigma']$. This is because neither strategy affects this edge.

```
  from y_notin_X edge σ_dom τ_dom
  have path_2: "path ?Gσ' y [y] z"
    unfolding induced_subgraph_def E_of_strat_def
    by (auto simp: map_add_dom_app_simps(3))
```

Because these paths make up the path from x to z, and both exist in $\mathcal{G}[\sigma']$, the whole path xs exists in $\mathcal{G}[\sigma']$. This completes the auxiliary lemma, stating that any nonempty path from x in $\mathcal{G}[\tau]$ that is restricted to t, which leads to a z in X, and does not intersect with X before that, also exists in $\mathcal{G}[\sigma']$. We name the lemma `path_no_X`.

```
  from path_1 path_2 xs_comp have "path ?Gσ' x xs z" by simp
} note path_no_X=this
```

Now, we use the fact that there is at least one escape from t, and the fact that all escapes lie in X to get an $\overline{\alpha}$-node y in t that has a successor z in X. This also means we have an edge from y to z in $\mathcal{G}[\sigma']$.

```
from <opponent_escapes t ≠ {}> <opponent_escapes t ⊆ X> obtain y z where
  y_opp_in_t: "y ∈ t ∩ V_α̅" and
  z_in_X: "z ∈ X" and
  y_z_edge: "(y,z) ∈ E"
  unfolding opponent_escapes_def by blast
with σ'_dom have y_z_edge_Gσ': "(y,z) ∈ ?Gσ'"
  by (simp add: ind_subgraph_notin_dom)
```

We obtain a path xs from x to y in $\mathcal{G}[\tau]$ restricted to t. We need to consider two cases: one where xs intersects with X, and one where it does not.

```
from y_opp_in_t x_in_t_min_X t_connected obtain xs where
  path_x_y: "path (Restr ?Gτ t) x xs y"
  using strongly_connected_path by blast
consider (X_in_xs) "set xs ∩ X ≠ {}" | (X_notin_xs) "set xs ∩ X = {}"
  by blast
thus ?thesis proof cases
```

In the case that xs intersects with X, we can obtain the subpath xs' of xs to the first y' in xs that is part of X. This path must always contain at least x, because x is not part of X, meaning our subpath xs' is not empty.

```
  case X_in_xs
  obtain xs' y' where
    y'_in_X: "y' ∈ X" and
    xs'_no_X: "set xs' ∩ X = {}" and
    path_x_y': "path (Restr ?Gτ t) x xs' y'"
    using shortest_subpath_to_region[OF path_x_y X_in_xs] by blast
  with x_in_t_min_X have "xs' ≠ []" by force
```

This is now a nonempty path from x to a node in X that exists within $\mathcal{G}[\tau]$ restricted to t, and does not intersect with X. By our auxiliary lemma **path_no_X**, this path also exists in $\mathcal{G}[\sigma']$. Using our earlier auxiliary lemma **append_path_to_A**, we now show that this can be combined with a path from y' to some node in A to form a path to A from x.

```
  from path_no_X[OF y'_in_X <xs≠[]> xs'_no_X path_x_y']
    y'_in_X X_path_to_A
  show ?thesis using append_path_to_A by blast
```

The case of X not being part of xs requires another case distinction. If xs is an empty path, then there exists a path from x directly to z, and we can use our earlier auxiliary lemma `append_path_to_A` to complete the proof of that case.

```
next
  case X_notin_xs
  show ?thesis proof (cases xs)
    case Nil
    with path_x_y y_z_edge_Gσ'
    have x_z_edge_Gσ': "path ?Gσ' x [x] z" by auto
    with z_in_X X_path_to_A show ?thesis
      using append_path_to_A by blast
```

If xs is not empty, we have to make yet another case distinction: one where y is part of X, and one where y is not part of X. In the former case, the original path qualifies as a path to X to which we can append a path to A.

```
  next
    case (Cons a list)
    consider (y_in_X) "y ∈ X" | (y_notin_X) "y ∉ X" by blast
    thus ?thesis proof cases
      case y_in_X
      with path_no_X[OF this _ X_notin_xs path_x_y]
           X_path_to_A Cons
      show ?thesis using append_path_to_A by blast
```

In the latter case, now know that y lies in $t \setminus X$. Furthermore, xs also lies in $t \setminus X$ because it does not intersect with X, and exists in $\mathcal{G}[\tau]$ restricted to t.

```
    next
      case y_notin_X
      with y_opp_in_t have "y ∈ t-X" by blast

      from restr_V_path[OF path_x_y] Cons X_notin_xs
      have "set xs ⊆ t-X" by blast
```

Our path from x to y exists entirely within $\mathcal{G}[\tau]$ and stays within $t \setminus X$, which means it is also a path in $\mathcal{G}[\sigma']$.

```
from path_x_y have "path ?Gτ x xs y"
  using path_inter(1) by fast
from path_restr_V[OF this <set xs⊆t-X> <y∈t-X>]
  subgraph_path[OF subgraph_τ]
have "path ?Gσ' x xs y" by simp
```

Now, we can append the edge from y to z to get a path from x to z. Because z is part of X, we can obtain a path from it to a node in A. We can combine the paths to get a path from x to a node in A.

```
  with <(y,z)∈Gσ'> have "path ?Gσ' x (xs@[y]) z" by simp
  with <z∈X> X_path_to_A show ?thesis
    using append_path_to_A by blast
qed
...
```

This completes our proof, showing that from each node in X', there exists a path in $\mathcal{G}[\sigma']$ to a node in A. □

We can now put all of our previous proofs together to show that our invariant is preserved at each step of our `tangle_attractor_step` relation.

Lemma 5.15. *If our invariant* `tangle_attractor_step_I` *(Listing 5.11) holds for the initial state* (X, σ)*, our* `tangle_attractor_step` *relation gives us a state* (X', σ') *for which the invariant also holds.*

Proof. The invariant requires seven properties to hold for X', σ'. Each of these properties corresponds to a lemma we have proven earlier in this section. The first property requires that A is a subset of X', which we can show is true because of the monotonic increase of X shown in Lemma 5.2. Because X contains A, and X is a subset of X', X' also contains A. The second property requires that σ' is a valid strategy for α, which follows from Lemma 5.6. The third property requires that the domain of σ' is all α-nodes in $X' \setminus A$, which follows from Lemma 5.7. The fourth property asks that the range of σ' lies within X', which was proven in Lemma 5.8. The fifth property states that, in $\mathcal{G}[\sigma']$, the region X' is partially closed in $X' \setminus A$. We proved this in Lemma 5.10. The sixth property needs all plays starting in X' to move to A or be won by α in $\mathcal{G}[\sigma']$. This was proven in Lemma 5.12. Finally, the seventh property requires that from every node in X', there exists a path to some node in A in $\mathcal{G}[\sigma']$. We showed this in Lemma 5.14. This means all seven properties are accounted for, completing our proof. □

Shown below (Listings 5.20 and 5.21) are this Isabelle proofs of this lemma and its equivalent for the reflexive transitive closure of the `tangle_attractor_step` relation. We simply use previous lemmas, much like in the proof above.

```
lemma tangle_attractor_step_preserves_I:
  "⟦tangle_attractor_step (X,σ) (X',σ'); tangle_attractor_step_I (X,σ)⟧
    ⟹ tangle_attractor_step_I (X',σ')"
  using tangle_attractor_step_mono[of X σ X' σ']
  using tangle_attractor_step_strat_of_Vα[of X σ X' σ']
  using tangle_attractor_step_dom[of X σ X' σ']
  using tangle_attractor_step_ran[of X σ X' σ']
  using tangle_attractor_step_closed_X'[of X σ X' σ']
  using tangle_attractor_step_forces_A_or_wins[of X σ X' σ']
  using tangle_attractor_step_path_to_A[of X σ X' σ']
  unfolding tangle_attractor_step_I_def split
  by auto
```

Listing 5.20: The tangle attractor step preserves our invariant.

```
lemma tangle_attractor_step_rtranclp_preserves_I:
  "⟦tangle_attractor_step** (X,σ) (X',σ'); tangle_attractor_step_I (X,σ)⟧
    ⟹ tangle_attractor_step_I (X',σ')"
  apply (induction rule: rtranclp_induct2)
  using tangle_attractor_step_preserves_I by blast+
```

Listing 5.21: The reflexive transitive closure of our step preserves our invariant.

Properties of Tangle Attractors

Now, we can use the invariant preservation we have proven to prove properties of the tangle attractor. Because we will need to use the invariant often, we define an auxiliary lemma (Listing 5.22). This lemma tells us that if we have a tangle attractor with region X and strategy σ, there exists a strategy σ' that satisfies the invariant with X, and that can be extended with `A_target X` to get σ. These properties will be used to relate between σ and σ'.

```
lemma player_tangle_attractor_I_aux:
  "player_tangle_attractor X σ
    ⟹ ∃σ'. tangle_attractor_step_I (X,σ') ∧ σ = σ' ++ A_target X"
  using tangle_attractor_step_rtranclp_preserves_I tangle_attractor_step_I_base
  unfolding player_tangle_attractor_def by blast
```

Listing 5.22: A tangle attracted region X with strategy σ has a σ' which satisfies our invariant and is extended with A_target to get σ.

With this lemma, we can start proving properties of the tangle attractor. We start with some fundamental properties.

Theorem 5.1. *The attracted region X of our tangle attractor contains the target region A.*

Proof. Our invariant includes the property that A is part of the region. Since we have shown this is true in the initial state of the tangle attractor (Lemma 5.5), and is preserved (Lemma 5.15), it is also true for the final state of the attractor. □

Our Isabelle proof uses the auxiliary lemma `tangle_attractor_step_I_aux`. It and the theorem for the `tangle_attractor` are shown in Listing 5.23 below.

```
lemma target_in_player_tangle_attractor:
  "player_tangle_attractor X σ ⟹ A ⊆ X"
  using player_tangle_attractor_I_aux[of X σ]
  unfolding tangle_attractor_step_I_def split
  by fast

theorem target_in_tangle_attractor:
  assumes "finite T"
  shows "tangle_attractor α T A X σ ⟹ A ⊆ X"
  using P0.target_in_player_tangle_attractor[of "{t∈T. tangle EVEN t}" A X σ]
  using P1.target_in_player_tangle_attractor[of "{t∈T. tangle ODD t}" A X σ]
  by (cases α; simp add: assms)
```

Listing 5.23: The tangle attractor always contains A.

Lemma 5.16. *Our tangle attractor is a subset of its target region A and V.*

Proof. We use our Lemma 5.3, which shows that each step's X' is a subset of $X \cup V$. Since our tangle attractor starts with an initial X that is equal to A, the final X contains A and nodes from V that were added by its steps. □

The Isabelle proof of this property is shown in Listing 5.24. We will use this property for the following theorems.

```
lemma player_tangle_attractor_ss:
  "player_tangle_attractor X σ ⟹ X ⊆ A ∪ V"
  unfolding player_tangle_attractor_def
  using tangle_attractor_step_rtranclp_ss by fastforce
```

Listing 5.24: The tangle attractor is a subset of target region A and V.

Theorem 5.2. *If the target region A is part of V, then our tangle-attracted region X is also part of V.*

Proof. By Lemma 5.16, the attracted region is a subset of A and V. Therefore, if A is a subset of V, the attracted region is entirely a subset of V. □

In Isabelle, our proof uses the same approach. The proofs for `player_tangle_attractor` and `tangle_attractor` are shown in Listing 5.25.

```
lemma player_tangle_attractor_in_V:
  "⟦player_tangle_attractor X σ; A ⊆ V⟧ ⟹ X ⊆ V"
  using player_tangle_attractor_ss by blast

theorem tangle_attractor_in_V:
  assumes "finite T"
  shows "⟦tangle_attractor α T A X σ; A ⊆ V⟧ ⟹ X ⊆ V"
  using P0.player_tangle_attractor_in_V[of "{t∈T. tangle EVEN t}" A X σ]
  using P1.player_tangle_attractor_in_V[of "{t∈T. tangle ODD t}" A X σ]
  by (cases α; auto simp: assms)
```

Listing 5.25: Our tangle attractor gives an attracted region in V.

Theorem 5.3. *If the target region A is finite, our tangle attractor gives a region X that is also finite.*

Proof. Because our tangle attractor is a subset of $A \cup V$ by Lemma 5.16, and V is finite, our attracted region is also finite. □

```
lemma player_tangle_attractor_finite:
  "⟦player_tangle_attractor X σ; finite A⟧ ⟹ finite X"
  using finite_subset[OF player_tangle_attractor_ss] by blast

theorem tangle_attractor_finite:
  assumes "finite T"
  shows "⟦tangle_attractor α T A X σ; finite A⟧ ⟹ finite X"
  using P0.player_tangle_attractor_finite[of "{t∈T. tangle EVEN t}" A X σ]
  using P1.player_tangle_attractor_finite[of "{t∈T. tangle ODD t}" A X σ]
  by (cases α; simp add: assms)
```

Listing 5.26: If the target region is finite, the tangle-attracted region is finite.

Now, we will prove that our strategy σ for the tangle attractor is a valid witness strategy for showing our definition is a valid tangle attractor. We do this in several lemmas, which we assemble to prove our final theorem.

Lemma 5.17. *The strategy σ of our tangle attractor is a valid strategy for α.*

Proof. Once again, this is a property of our invariant. It holds for the initial state of our attractor (Lemma 5.5), and is preserved at each step (Lemma 5.15), so it holds for the σ' obtained at the end. The strategy given by `A_target X` is also a valid strategy, and combining the two gives a strategy that is also valid for α. □

Our Isabelle proof uses the auxiliary lemma `player_tangle_attractor_I_aux`. It also uses our lemma showing that `A_target X` gives a valid strategy for α. The proof is shown in Listing 5.27.

```
lemma player_tangle_attractor_strat_of_Vα:
  "player_tangle_attractor X σ ⟹ strategy_of Vα σ"
  using player_tangle_attractor_I_aux[of X σ] A_target_strat[of X]
  unfolding tangle_attractor_step_I_def split
  by fastforce
```

Listing 5.27: The tangle attractor's strategy σ is a valid strategy for α.

Lemma 5.18. *The strategy of our tangle attractor has only α-nodes in X in its domain.*

Proof. By our invariant, the obtained σ' of our attractor's step has in its domain all α-nodes in $X \setminus A$. The domain of `A_target X` includes some number of α-nodes in A (possibly none). Combining the two gives a strategy with as its domain some subset of all α-nodes in X. □

Our Isabelle proof (Listing 5.28) uses our auxiliary lemma for the invariant, and a lemma giving the domain of `A_target X`.

```
lemma player_tangle_attractor_strat_dom:
  "player_tangle_attractor X σ ⟹ dom σ ⊆ Vα ∩ X"
  using player_tangle_attractor_I_aux[of X σ] A_target_dom[of X]
  unfolding tangle_attractor_step_I_def split
  by auto
```

Listing 5.28: The tangle attractor's strategy σ has its domain within $V_\alpha \cap X$.

Lemma 5.19. *The range of the σ of our tangle attractor lies in X.*

Proof. By our invariant, the range of the obtained σ' from our steps lies in X. The range of `A_target X` also lies in X. Consequently, the combination of the two also lies in X. □

Our Isabelle proof is much the same as the previous, shown in Listing 5.29.

```
lemma player_tangle_attractor_strat_ran:
  "player_tangle_attractor X σ ⟹ ran σ ⊆ X"
  using player_tangle_attractor_I_aux[of X σ] A_target_ran[of X]
  unfolding tangle_attractor_step_I_def split
  by (auto simp: ran_def)
```

Listing 5.29: The range of our tangle attractor strategy lies within the attracted region.

Lemma 5.20. *In the subgame $\mathcal{G}[\sigma]$ of the tangle attractor's strategy σ, the attracted region X is partially closed in $X \setminus A$.*

Proof. In the subgame $\mathcal{G}[\sigma']$ for the σ' obtained by our attractor's steps, the region X is partially closed in $X \setminus A$ by our invariant. Since our strategy `A_target X` only assigns successors to nodes in A, it does not affect this partial closedness. Therefore, the combined strategy σ is also partially closed in $X \setminus A$. □

Our Isabelle proof (Listing 5.30) once again uses our auxiliary lemma. In the `clarsimp` step, it gets the σ' obtained by our `tangle_attractor_step`. We then use this with the lemma `add_A_target_A_notin_dom`, which shows that the combined strategy gives a subgame that is a subgraph of the subgame of the original σ'.

```
lemma player_tangle_attractor_strat_partially_closed:
  "player_tangle_attractor X σ ⟹ induced_subgraph σ `` (X-A) ⊆ X"
  using player_tangle_attractor_I_aux[of X σ]
  unfolding tangle_attractor_step_I_def split
  apply clarsimp
  subgoal for x y σ'
    using add_A_target_A_notin_dom(1)[of σ' X] by blast
  done
```

Listing 5.30: In the subgame of the tangle attractor's strategy, the attracted region X is partially closed in $X \setminus A$.

Lemma 5.21. *Our tangle attractor's strategy σ forces every play starting in the attracted region X to either move to the target region A or be won by α in the subgame $\mathcal{G}[\sigma]$.*

Proof. The σ' obtained by our `tangle_attractor_step` forces all plays starting in the attracted region X to move to A in the subgame $\mathcal{G}[\sigma']$ by our invariant. Our strategy `A_target X` only assigns moves to α-nodes in A. This means that plays that do not move to A are entirely unaffected in the combined strategy σ. Plays that do move to A will still move to A, as they will have the same moves leading up to it. Therefore, in the subgame $\mathcal{G}[\sigma]$ of σ, all plays either move to A, or are won by α. □

Our Isabelle proof is shown in Listing 5.31. It is similar to the previous proof, using our auxiliary lemma, as well as the lemma `add_A_target_A_notin_dom`. It additionally uses `subgraph_lasso` to show that any lassos in the subgame $\mathcal{G}[\sigma]$ also existed in the original $\mathcal{G}[\sigma']$.

```
lemma player_tangle_attractor_strat_forces_A_or_wins:
  "player_tangle_attractor X σ ⟹ ∀x∈X. ∀xs ys. lasso (induced_subgraph σ) x xs ys
    ⟶ set (xs@ys) ∩ A ≠ {} ∨ winning_player ys"
  using player_tangle_attractor_I_aux[of X σ]
  unfolding tangle_attractor_step_I_def split
  apply clarsimp
  subgoal for x xs ys σ'
    using subgraph_lasso[OF add_A_target_A_notin_dom(1)[of σ' X]]
    by blast
  done
```

Listing 5.31: The tangle attractor's strategy forces every play in its subgame that starts in X to either move to A or be won by α.

Lemma 5.22. *In the subgame $\mathcal{G}[\sigma]$ of the strategy σ obtained by our tangle attractor, from every node in the attracted region X, there exists a path to some node in the target region A.*

Proof. We will follow along with our Isabelle proof again. This starts by setting the assumption that we have an attracted region X with attractor strategy σ, and a node x in X. We must now show that there exists a path from this x to some node in A.

We obtain the strategy σ' that was created through our `tangle_attractor_step` relation, for which we know that its domain is all α-nodes in $X \setminus A$, that we can obtain a path in $\mathcal{G}[\sigma']$ from any node in X to a node in A, and that σ is created by combining σ' with `A_target X`. Then, we define shorthand for $\mathcal{G}[\sigma]$ and $\mathcal{G}[\sigma']$.

```
lemma player_tangle_attractor_strat_path_to_A:
  "player_tangle_attractor X σ ⟹
    ∀x∈X. ∃y∈A. ∃xs. path (induced_subgraph σ) x xs y"
proof (rule ballI)
  fix x assume attr: "player_tangle_attractor X σ" and x_in_X: "x ∈ X"
  from player_tangle_attractor_I_aux[OF attr] obtain σ' where
    σ'_dom: "dom σ' = V_α ∩ (X-A)" and
    σ'_path_to_A: "∀x∈X. ∃y∈A. ∃xs. path (induced_subgraph σ') x xs y" and
    σ_comp: "σ = σ' ++ A_target X"
    unfolding tangle_attractor_step_I_def split
    by blast
  let ?Gσ = "induced_subgraph σ"
  let ?Gσ' = "induced_subgraph σ'"
```

Now, we look at two cases. One where x is a node in A, and one where x is not a node in A. In the former case, we have an empty path from x to itself, which satisfies our condition, thus showing the thesis.

```
  consider (x_in_A) "x∈A" | (x_notin_A) "x∉A" by blast
  thus "∃y∈A. ∃xs. path ?Gσ x xs y" proof cases
    case x_in_A show ?thesis
      apply (rule bexI[where x=x])
      apply (rule exI[where x="[]"])
      using x_in_A by auto
```

In the case where x is not a node in A, we get a path in $\mathcal{G}[\sigma']$ from x to some node in A, and from that get the shortest subpath xs to a y in A, where xs does not intersect with A. Consequently, the path xs is nonempty, because x is not a node in A and y is a node in A, so there must be at least one step in the path.

```
next
  case x_notin_A
  from x_in_X σ'_path_to_A obtain xs y where
    y_in_A: "y∈A" and
    xs_no_A: "set xs ⊆ -A" and
    path: "path ?Gσ' x xs y"
    using shortest_subpath_to_region[of ?Gσ' x]
    by (metis inf_shunt)
  from path xs_no_A <x∉A> <y∈A> have "xs ≠ []" by force
```

We now get the portion of the path xs leading to the node before y, xs' to y'. This entire path is now outside of A, and the destination y' is also outside of A.

```
  from path obtain xs' y' where
    edge: "(x,y) ∈ ?Gσ'" and
    xs_comp: "xs = xs'@[y]" and
    path': "path ?Gσ' x xs' y'"
    using path_D_rev[OF path <xs≠[]>] by blast
  with xs_no_A have "set xs' ⊆ -A" "y' ∈ -A" by auto
```

Because this path is outside A, it is not affected by `A_target X`, so it must also exist in $\mathcal{G}[\sigma]$.

```
  from path_restr_V[OF path' this] σ'_dom σ_comp
  have "path ?Gσ x xs' y'"
    using subgraph_path[OF add_A_target_A_notin_dom(2)[of σ' X]]
    by blast
```

Moreover, the edge from y' to y also exists in $\mathcal{G}[\sigma]$, because it lies outside A, thus it is not affected by `A_target X`. Isabelle needs us to unfold some definitions and needs an auxiliary lemma about adding maps to deduce this.

```
  moreover from edge <y'∈-A> σ_comp have "(y',y) ∈ ?Gσ"
    unfolding induced_subgraph_def E_of_strat_def A_target_def
    by (auto simp: map_add_Some_iff)
```

Together, the path xs' and the edge from y' to y form the original path xs from x to y. We now see that it exists in the subgame $\mathcal{G}[\sigma]$, as both xs' and the edge exist in this subgame. Because y is a node in A, the path now satisfies our thesis for this case.

```
  ultimately have "path ?Gσ x xs y"
    using xs_comp by auto

  with <y∈A> show ?thesis by blast
 qed
qed
```

With this, we have completed our proof that we can obtain a path in $\mathcal{G}[\sigma]$ from any node in the attracted region X to a node in the target region A. □

We now state and prove our theorem for the properties of the attractor strategy σ.

Theorem 5.4. *When we have a tangle attracted region X for player α, with the tangle strategy σ, this strategy exhibits the following properties:*

1. *σ is a valid strategy for α*
2. *The domain of σ contains only α-nodes in X*
3. *The range of σ lies in X*
4. *X is partially closed in $X \setminus A$ in the subgame $\mathcal{G}[\sigma]$*

5. *All plays in* $\mathcal{G}[\sigma]$ *that start in* X *either move to* A *or are won by* α
6. *From all nodes in* X*, there exists a path in* $\mathcal{G}[\sigma]$ *to some node in* A*.*

Proof. We use the previous lemmas to prove each of these properties. Property 1 is shown by Lemma 5.17, property 2 is shown by Lemma 5.18, property 3 is shown by Lemma 5.19, property 4 is shown by Lemma 5.20, property 5 is shown by Lemma 5.21, and finally, property 6 is shown by Lemma 5.22. □

In Isabelle, our proof is much the same. The proofs for the `player_tangle_attractor` and the `tangle_attractor` are both shown in Listing 5.32.

```
theorem player_tangle_attractor_strat:
  "player_tangle_attractor X σ ⟹
    strategy_of V_α σ ∧
    dom σ ⊆ V_α ∩ X ∧
    ran σ ⊆ X ∧
    induced_subgraph σ `` (X-A) ⊆ X ∧
    (∀x∈X. ∀xs ys. lasso (induced_subgraph σ) x xs ys
      ⟶ set (xs@ys) ∩ A ≠ {} ∨ winning_player ys) ∧
    (∀x∈X. ∃y∈A. ∃xs. path (induced_subgraph σ) x xs y)"
  using player_tangle_attractor_strat_of_V_α[of X σ]
    player_tangle_attractor_strat_dom[of X σ]
    player_tangle_attractor_strat_ran[of X σ]
    player_tangle_attractor_strat_partially_closed[of X σ]
    player_tangle_attractor_strat_forces_A_or_wins[of X σ]
    player_tangle_attractor_strat_path_to_A[of X σ]
  by fast

theorem tangle_attractor_strat:
  assumes "finite T"
  shows "tangle_attractor α T A X σ ⟹
    strategy_of_player α σ ∧
    dom σ ⊆ V_player α ∩ X ∧
    ran σ ⊆ X ∧
    induced_subgraph σ `` (X-A) ⊆ X ∧
    (∀x∈X. ∀xs ys. lasso (induced_subgraph σ) x xs ys
      ⟶ set (xs@ys) ∩ A ≠ {} ∨ player_wins_list α ys) ∧
    (∀x∈X. ∃y∈A. ∃xs. path (induced_subgraph σ) x xs y)"
  unfolding strategy_of_player_def
  using P0.player_tangle_attractor_strat[of "{t∈T. tangle EVEN t}" A X σ]
  using P1.player_tangle_attractor_strat[of "{t∈T. tangle ODD t}" A X σ]
  by (cases α; simp add: assms V_α_def)
```

Listing 5.32: Proofs of the properties of the tangle attractor's strategy.

5.2.2 Termination Proof

We did not need to prove termination for our inductive set that constructed a regular attractor, which constructs a maximal set automatically. Our definition of a tangle attractor uses an inductive predicate for a relation instead. This construction does not terminate in the same way and Isabelle does not prove its termination automatically. This means we have to prove termination ourselves. We achieve this by proving that our step relation is well-founded, meaning that some element of it is strictly decreasing with each step.

Lemma 5.23. *Our* `tangle_attractor_step` *is a well-founded relation.*

Proof. In our `tangle_attractor_step` relation, the set of remaining nodes outside of the attractor, $V \setminus X$, decreases with every step, and eventually reaches an empty set. The former is true because at each step, the relation adds at least one node that was not previously part of the attractor. This

is either a α-node with a successor in X, an $\overline{\alpha}$-node with only successors in X, or a node from a tangle that $\overline{\alpha}$ can only escape from by moving to X. Because V is finite, this can only continue for a limited number of steps, before no more nodes exist outside the attractor, and it cannot be extended further. □

Isabelle has the `wfP` function for showing that predicates are well-founded. In our scenario, Isabelle's definition for this function is counter to our intuition: it needs to be shown for the reverse of the relation, working backwards from a final state rather than forwards from an initial state. Listing 5.33 shows our Isabelle proof using this function. Here, the reverse of the relation is denoted with the superscript $^{-1-1}$.

```
lemma tangle_attractor_step_wf: "wfP (tangle_attractor_step⁻¹⁻¹)"
  unfolding wfP_def
  apply (rule wf_subset[of "inv_image (finite_psubset) (λ(s,σ). V-s)"]; clarsimp)
  apply (erule tangle_attractor_step.cases)
  subgoal using V_α_subset by blast
  subgoal by auto
  subgoal using player_tangle_in_V tangles_T by blast
  done
```

Listing 5.33: The step relation of a tangle attractor is well-founded.

Because our definition works by taking a step in the reflexive transitive closure of the relation, where the result of that step is not in the domain of any further steps, we need to show that we can get this from our well-founded relation. We define a lemma (Listing 5.34) that says we can get a final state for any well-founded relation.

```
lemma wf_rel_terminates:
  "wfP R⁻¹⁻¹ ⟹ ∃X' σ'. R** S (X',σ') ∧ ¬Domainp R (X',σ')"
  unfolding wfP_def
  apply (induction S rule: wf_induct_rule)
  subgoal for x apply (cases "Domainp R x"; clarsimp)
    subgoal using converse_rtranclp_into_rtranclp[of R] by blast
    subgoal using surj_pair[of x] by blast
    done
  done
```

Listing 5.34: A well-founded relation will always yield a final state.

Finally, we can use this lemma to prove that there will always exist a tangle attractor X with corresponding strategy σ for the target set A. The proofs for both `player_tangle_attractor` and `tangle_attractor` can be seen below in Listing 5.35.

```
lemma player_tangle_attractor_exists:
  "∃X σ. player_tangle_attractor X σ"
  using wf_rel_terminates[OF tangle_attractor_step_wf]
  unfolding player_tangle_attractor_def by simp

theorem tangle_attractor_exists:
  assumes "finite T"
  shows "∃X σ. tangle_attractor α T A X σ"
  using P0.player_tangle_attractor_exists P1.player_tangle_attractor_exists
  by (cases α; simp add: assms)
```

Listing 5.35: There always exists a tangle attractor.

5.2.3 Further Properties

There are a few additional properties we prove for tangle attractors. These are not required for correctness of the tangle attractor itself, but they will be used in later proofs. We will cover these here.

Lemma 5.24. *If we have a tangle-attracted region X with strategy σ for player α, and we have a α-node x in A that has a successor in X, then x is part of the domain of σ.*

Proof. Any α-node in A that has a successor in X is part of the domain of `A_target`, by its definition. Therefore, this node is part of the domain of σ, by the definition of `player_tangle_attractor`. □

The proofs for this property in both locales are automated proofs, shown in Listing 5.36.

```
lemma player_tangle_attractor_in_dom_A:
  "⟦player_tangle_attractor X σ; x∈V_α ∩ A; E `` {x} ∩ X ≠ {}⟧ ⟹ x ∈ dom σ"
  using player_tangle_attractor_I_aux[of X σ] A_target_in_dom[of x X]
  unfolding tangle_attractor_step_I_def split
  by fastforce

lemma tangle_attractor_in_dom_A:
  assumes "finite T"
  shows "⟦tangle_attractor α T A X σ; x ∈ V_player α ∩ A; E `` {x} ∩ X ≠ {}⟧
    ⟹ x ∈ dom σ"
  using P0.player_tangle_attractor_strat_in_dom_A[of "{t∈T. tangle EVEN t}" A X σ]
  using P1.player_tangle_attractor_strat_in_dom_A[of "{t∈T. tangle ODD t}" A X σ]
  by (cases α; simp add: assms)
```

Listing 5.36: α-nodes in A with a successor in tangle-attracted region X, are part of the domain of witness strategy σ.

Lemma 5.25. *If we have a tangle-attracted region X with strategy σ for player α, and we have a α-node x in $X \setminus A$, then x is part of the domain of σ.*

Proof. Any α-node in $X \setminus A$ is part of the domain of the σ' obtained by our `tangle_attractor_step` relation. By the definition of `player_tangle_attractor`, the node is therefore part of the domain of σ. □

Proofs for both versions of this lemma are shown in Listing 5.37.

```
lemma player_tangle_attractor_in_dom_not_A:
  "⟦player_tangle_attractor X σ; x∈V_α ∩ (X-A)⟧ ⟹ x ∈ dom σ"
  using player_tangle_attractor_I_aux[of X σ]
  unfolding tangle_attractor_step_I_def split
  by (clarsimp simp: domD)

lemma tangle_attractor_in_dom_not_A:
  assumes "finite T"
  shows "⟦tangle_attractor α T A X σ; x ∈ V_player α ∩ (X-A)⟧ ⟹ x ∈ dom σ"
  using P0.player_tangle_attractor_strat_in_dom_not_A
    [of "{t∈T. tangle EVEN t}" A X σ]
  using P1.player_tangle_attractor_strat_in_dom_not_A
    [of "{t∈T. tangle ODD t}" A X σ]
  by (cases α; simp add: assms)
```

Listing 5.37: α-nodes in tangle-attracted region X, but not in A, are part of the domain of witness strategy σ.

Lemma 5.26. *All nodes outside a tangle-attracted region have successors outside that region.*

Proof. Consider a node v outside of X. If we do a proof by contradiction, all successors of this node are part of X. If this node is a α-node, it would have been attracted through the `player` step, as it is a α-node that can move to X. If it is an $\overline{\alpha}$-node, it would have been attracted through the `opponent` step, because it is an $\overline{\alpha}$-node with only successors in X. Both cases give us a contradiction, so this node must have successors outside X. □

Our Isabelle proofs for both the `player_tangle_attractor` and the `tangle_attractor` are short automated proofs that use a similar structure to the former proof. These proofs are shown in Listing 5.38.

```
lemma notin_player_tangle_attractor_succ:
  "⟦player_tangle_attractor X σ; v ∈ V; v ∉ V⟧ ⟹ E `` {v} - X ≠ {}"
  unfolding player_tangle_attractor_def
  using succ[of v] apply (cases "v∈V_α")
  by (auto intro: tangle_attractor_step.intros)

lemma notin_tangle_attractor_succ:
  assumes "finite T"
  shows "⟦tangle_attractor α T A X σ; v ∈ V; v ∉ X⟧ ⟹ E `` {v} - X ≠ {}"
  using P0.notin_player_tangle_attractor_succ[of "{t∈T. tangle EVEN t}" A X σ]
  using P1.notin_player_tangle_attractor_succ[of "{t∈T. tangle ODD t}" A X σ]
  by (cases α; simp add: assms)
```

Listing 5.38: All nodes outside a tangle-attracted region X have successors outside X.

We use the prior lemma to show that we get a valid subgame when we remove an attractor from the game. This means the subgame satisfies all the assumptions we gave in our locales. Most importantly, this game needs to have no dead ends, and its edges must be a subset of its nodes.

Lemma 5.27. *When we remove a tangle attractor from the game, the remainder of the game is a valid subgame.*

Proof. By our previous Lemma 5.26, all nodes outside the tangle attractor have successors that are not part of the attractor. This means that removing it gives a subgame without dead ends, making it a valid subgame. □

In Isabelle, our proof (Listing 5.39) is straightforward, using `unfold_locales` to expand the requirements of the locale `paritygame`, and our previous lemma to show that there are no dead ends.

```
lemma remove_tangle_attractor_subgame:
  assumes "finite T"
  assumes "tangle_attractor α T A X σ"
  shows "valid_subgame (V-X)"
  apply simp
  apply (unfold_locales; clarsimp)
  using notin_tangle_attractor_succ[OF assms] E_in_V by blast
```

Listing 5.39: Removing a tangle attractor from the game gives a valid subgame.

5.2.4 α-Maximality

With tangle attractors defined, we continue to define α-maximal regions. Recall from definition 3.9 that a region is α-maximal when $TAttr_\alpha^{\mathcal{G},T}(A) = A$. This means we can no longer extend it using a tangle attractor. We define this in Isabelle in the `player_paritygame` locale, saying that a region A is α-maximal for player α if for every possible starting strategy σ, there does not exist a state S' that is a valid attractor step from the state (A, σ). As this definition relies on tangle attractors, it also needs to be given a filtered set of tangles in its definition in the locale `paritygame`. The two definitions are shown in Listing 5.40.

```
definition player_α_max :: "'v set ⇒ bool" where
  "player_α_max ≡ ∀σ. ∄S'. tangle_attractor_step A (A,σ) S'"

fun α_max :: "player ⇒ 'v set set ⇒ 'v set ⇒ bool" where
  "α_max EVEN T = P0.player_α_max {t∈T. tangle EVEN t}"
| "α_max ODD T = P1.player_α_max {t∈T. tangle ODD t}"
```

Listing 5.40: α-maximality defined in Isabelle.

We prove that any region X obtained with a tangle attractor is α-maximal. This is a simple automated proof in Isabelle, which is shown for both versions of our definition in Listing 5.41.

```
lemma player_tangle_attractor_is_α_max:
  "player_tangle_attractor A X σ ⟹ player_α_max X"
  unfolding player_tangle_attractor_def player_α_max_def
  by (auto simp: tangle_attractor_step.simps intro: tangle_attractor_step.intros)

lemma tangle_attractor_is_α_max:
  assumes "finite T"
  shows "tangle_attractor α T A X σ ⟹ α_max α T X"
  using P0.player_tangle_attractor_is_α_max[of "{t∈T. tangle EVEN t}" A X σ]
  using P1.player_tangle_attractor_is_α_max[of "{t∈T. tangle ODD t}" A X σ]
  by (cases α; simp add: assms)
```

Listing 5.41: A tangle attractor for a player is α-maximal for that player.

In Van Dijk's proof for his third lemma [7, Section 4.5], this property is used to show that every tangle found by `Search` is one that was not found previously. We would likely require auxiliary lemmas about the implications of α-maximality if we were to show this property. At present, however, no such lemmas have been formulated because this part of the correctness proof has not yet been reached.

5.3 The Search Algorithm

Our formalisation of the `search` algorithm is based on the pseudocode of Algorithm 3.2. We define this formalisation and all of its proofs in the file `TangleLearning_Search.thy`. The main component of this algorithm is a while-loop, which terminates when the remaining region R in the game is empty. We formalise the whole algorithm as steps of this loop.

In this loop, two elements are actually changed by the algorithm, being the set of discovered tangles Y, and the remaining region R. The state of our algorithm, then, is the pair of (R, Y). We define a `type_synonym` for this state for the sake of readability, shown in Listing 5.42.

```
type_synonym 'a search_state = "'a set × 'a set set"
```

Listing 5.42: The state of the search algorithm.

We open an instance of the locale `paritygame` to house our definitions and proofs. We then open a context where we fix a T that is a finite set of tangles for one of the two players. We also assume that all tangles in this set are open, i.e., that the opponent can still leave these tangles. This is because closed tangles are not stored in T; they are dominions, and are instead used to get the winning regions in `solve`.

```
context paritygame begin
context
  fixes T :: "'v set set"
  assumes fin_T: "finite T"
  assumes tangles_T: "∀t∈T. tangle EVEN t ∨ tangle ODD t"
  assumes open_tangles_T:
    "∀t∈T. tangle α t ⟶ E `` (t ∩ V_opponent α) - t ≠ {}"
begin
```

Listing 5.43: The context for our definition of search.

We define the step relation of our loop as an inductive predicate (Listing 5.44). The `step` case uses as its first precondition that R must not be empty, which is the while-condition for the loop. Every subsequent precondition corresponds to a line in the pseudocode. It starts by saying p is the highest priority in R, and α is the player who wins p. Next, A is the set of all nodes in R that have priority p. Now, (Z, σ) is the tangle attractor for α to A in the subgraph R, using the tangles in T. The set of open top nodes Ov[1] then consists of α-nodes that have no successors in Z, and the $\overline{\alpha}$-nodes that have

[1] In the pseudocode, this variable is named O. Here, it is named Ov because the name O is already used in the Isabelle/HOL standard library.

successors outside Z. Finally, we have the preconditions that actually relate to the resulting state. If there are no open top nodes, then Y' should be Y plus all S that are nontrivial bottom SCCs that exist in Z in the induced subgraph of σ; otherwise, it is the same as Y. Furthermore, R' must be $R \setminus Z$.

```
inductive search_step
  :: "'v search_state ⇒ 'v search_state ⇒ bool"
where step:
  "⟦R ≠ {};
   p = pr_set R; α = player_wins_pr p;
   A = {v. v ∈ R ∧ pr v = p};
   subgraph_tattr R α T A Z σ;
   Ov = {v ∈ V_player α ∩ A. E `` {v} ∩ Z = {}}
      ∪ {v ∈ V_opponent α ∩ A. ¬E `` {v} ∩ R ⊆ Z};
   Y' = (if Ov = {} then Y ∪ {S. bound_nt_bottom_SCC Z σ S} else Y);
   R' = R-Z⟧
    ⟹ search_step (R,Y) (R',Y')"
```

Listing 5.44: The inductive predicate for each step of search.

Like with `tangle_attractor_step`, the induction rule for `search_step` does not let us specify the individual variables in our state. We once again define an induction rule `search_step_induct` that allows us to use (R, Y) as a state, rather than naming it S. This is the rule we use in further proofs.

Now, we give our complete definition for `search` in Listing 5.45. Our approach is similar to the one used in our definition for `player_tangle_attractor`, using the reflexive transitive closure of the `search_step` relation. We take an R for our initial state, and say that a given Y is the resulting set of tangles if we can reach $(\emptyset, Y)$ from $(R, \emptyset)$ with the step relation. We no longer have to specify that the resulting state is not in the domain of the relation; because the algorithm terminates when $R = \emptyset$, $(\emptyset, Y)$ is clearly not in its domain.

```
definition search :: "'v set ⇒ 'v set set ⇒ bool" where
  "search R Y ≡ search_step** (R,{}) ({},Y)"
```

Listing 5.45: Our definition of search.

5.3.1 Correctness Proof

To prove that `search` is correct, we need an invariant. If our invariant holds, it should imply the desired properties of Y when the algorithm terminates. First of all, Y should be finite. This may seem rather simple, but further proofs may not work if Y might be infinite. Second of all, Y should not be empty. We expect that `search` finds at least one tangle. Furthermore, since the purpose of `search` is finding new tangles, we want Y to contain tangles which are not already known.

We define an invariant (Listing 5.46) that should imply the given properties. Additionally, it contains properties that are necessary to prove the others. This is defined as a lambda function, which takes as input the state of the algorithm in the form of the current region R and the discovered set of tangles Y.

```
definition search_I :: "'v search_state ⇒ bool" where
  "search_I ≡ λ(R,Y). valid_subgame R ∧ finite Y ∧ (∀U ∈ Y. ∃α. tangle α U)"
```

Listing 5.46: Our invariant for the search algorithm.

Currently, this invariant does not prove all of the aforementioned properties. When defining an invariant, we start with a weak invariant we can prove, and make it stricter until it shows all desired properties. We have not yet been able to show that the tangles in Y were previously unknown. Therefore, this property is currently missing from the invariant. Also note that the fact that the algorithm finds at least one tangle is not included in the invariant. This property is not shown as part of the invariant, because it relies on the initial R given to `search`, which the invariant does not account for. We will show this property in Section 5.3.1 instead.

We will now show that this invariant yields the desired properties. This too is currently incomplete, missing the property that Y contains tangles that were previously unknown.

Lemma 5.28. *If our invariant* `search_I` *holds for a final state* $(\emptyset, Y)$*, then* Y *is a finite set of tangles.*

Clearly, this is trivially true, as these two properties are included in the definition of our invariant `search_I`. Our Isabelle proof for this property is as follows:

```
lemma search_I_correct:
  "search_I ({},Y) ⟹ finite Y ∧ (∀U ∈ Y. ∃α. tangle α U)"
  unfolding search_I_def split by fast
```

Invariant Preservation

Now, we will show that this invariant is preserved by our `search_step` relation. First, we show that our invariant holds for the initial state.

Lemma 5.29. *If* R *is a valid subgame, our invariant holds for the initial state* $(R, \emptyset)$ *in* `search`.

Proof. Because R is a valid subgame, the first property in the invariant is clearly true. The second property is that Y must be finite, which is trivially true because it is the empty set in the initial state. Finally, the property that all elements in Y are tangles is once again trivially true because Y is the empty set. □

The Isabelle proof for this property is shown in Listing 5.47 below.

```
lemma search_I_base: "valid_subgame R ⟹ search_I (R,{})"
  unfolding search_I_def split by blast
```

Listing 5.47: If R is a valid subgame, our invariant holds for the initial state of the search algorithm.

Next, we will show that this invariant is preserved by each step of the algorithm, and eventually at the end of `search`. We do this by proving the preservation of the individual properties separately. Afterwards, we will combine these proofs to show the invariant is preserved.

Lemma 5.30. *If the initial region* R *is a valid subgame, the resulting* R' *from our* `search_step` *relation is also a valid subgame.*

Proof. We will follow our Isabelle proof. This proof is an induction using our `search_step_induct` rule.

```
lemma search_step_valid_subgame:
  "⟦search_step (R,Y) (R',Y'); valid_subgame R⟧ ⟹ valid_subgame R'"
  case (step R p α A Z σ Ov Y' Y R')
```

From our `step` case, we know that Z is an attractor to A in the subgame R. We also know that R is a subgame, so we use `interpret` to let us use lemmas in the context of R as a parity game.

```
  hence tattr: "subgraph_tattr R α T A Z σ" by blast
  from <valid_subgame R> interpret R_game: paritygame "Restr E R" "V∩R" "Vα∩R" ..
```

Because R is a valid subgame, R is part of V, and because R' is obtained by removing Z from R, R' is also a subset of V.

```
  from <R'=R-Z> <valid_subgame R> have "R' ⊆ V" by auto
```

Furthermore, restricting the game to R' gives a valid parity game. This is due to Lemma 5.27, and the fact that we obtain R' by removing the attractor Z from R. Because the intersections in this property do not exactly match those in the definition of `valid_subgame`, we need to pass some additional lemmas to `simp`.

```
  moreover from <R'⊆V> <R'=R-Z> have "paritygame (Restr E' R) (V∩R') (V0∩R')"
    using R_game.remove_tangle_attractor_subgame[OF fin_T tattr]
    by (simp add: Times_Int_Times Int_assoc Int_absorb1 Int_Diff)
```

The former two properties now show that R' is a valid subgame: it is a subset of V that is a valid parity game.

```
  ultimately show ?case ..
qed
```

With this, we have completed our proof. □

Lemma 5.31. *If our invariant holds for the initial state* (R, Y)*, then our* `search_step` *relation obtains a* Y' *that is finite.*

Proof. By our invariant, the initial Y is finite. When Y' is obtained, it is either an unchanged Y, in which case it trivially remains finite, or it is Y extended with all nontrivial bottom SCCs in the attracted region Z. Z itself is finite per Theorem 5.3, so it may only contain a finite number of SCCs. Therefore, Y' should also be finite in the latter case. □

Our Isabelle proof (Listing 5.48) follows the same reasoning, using Theorem 5.3 and an induction using our induction rule for the step.

```
lemma search_step_finite_Y:
  "⟦search_step (R,Y) (R',Y'); search_I (R,Y)⟧ ⟹ finite Y'"
  unfolding search_I_def split
  apply (induction rule: search_step_induct)
  using paritygame.tangle_attractor_finite[OF _ fin_T]
  using finite_subset[OF _ fin_V] by force
```

Listing 5.48: If our invariant holds in the initial state, our step relation yields a finite Y'.

The next lemma we prove is Van Dijk's Lemma 2 [7, Section 4.5]. We will use this to prove that `search` finds tangles.

Lemma 5.32. *In a step in the* `search` *algorithm, all games that stay in the attracted region* Z *are won by* α*.*

Proof. We follow our Isabelle proof. This begins with the relevant preconditions that hold in a step of `search`. We have a valid subgame R; the top priority in R, p; the set of all nodes of priority p, A; and an attracted region Z to A for α in the subgame R with strategy σ, using the tangles in T. We show that for all nodes x in the region Z, every lasso from x with spoke xs and loop ys has ys won by α.

```
lemma search_step_games_in_Z_won:
  assumes valid_subgame_R: "valid_subgame R"
  assumes p_def: "p = pr_set R"
  assumes α_def: "α = player_wins_pr p"
  assumes A_def: "A = {v ∈ R. pr v = p}"
  assumes attr: "subgraph_tattr R α T A Z σ"
  shows "∀x∈Z. ∀xs ys. lasso (Restr (induced_subgraph σ) Z) x xs ys
    ⟶ player_wins_list α ys"
proof (intro ballI allI impI)
```

We begin our proof by using `interpret` to let use use lemmas and definitions in the context of the subgame R. Furthermore, R is a subset of V.

```
  from valid_subgame_R interpret R_game:
    paritygame "Restr E R" "V∩R" "V₀∩R" by blast
  from valid_subgame_R have "R⊆V" by simp
```

We define shorthand for $\mathcal{G}[\sigma]$, and use an auxiliary lemma to show that $\mathcal{G}[\sigma]$ in the the subgame R is equal to $\mathcal{G}[\sigma]$ in the whole game, restricted to R.

```
  let ?Gσ = "induced_subgraph σ"
  have R_subgraph_eq: "R_game.induced_subgraph σ = Restr ?Gσ R"
    using restr_ind_subgraph[OF R_game.arena_axioms] .
```

Now, we fix our x in Z and a lasso xs ys that stays in Z in $\mathcal{G}[\sigma]$. This means the loop ys is entirely part of Z, and our lasso exists within $\mathcal{G}[\sigma]$, without the restriction to Z. We use an auxiliary lemma for this. Furthermore, because R is a subset of V, ys contains only nodes in R.

```
  from restr_V_lasso[OF lasso] have
    ys_in_Z: "set ys ⊆ Z" and
    lasso_σ: "lasso ?Gσ x xs ys"
    by blast+
  with <R⊆V> have ys_in_R: "set ys ⊆ R"
    using R_game.tangle_attractor_in_T[OF fin_T attr]
    unfolding A_def by blast
```

We only have to reason about the loop ys, so we get the node y at the start of the loop, which is part of ys. We also state the fact that ys is non-empty, as the loop of a lasso is never empty. y also lies in Z, because ys contains only nodes in Z. With these, we have all the facts we will need for the rest of the proof.

```
  from lasso_loop[OF lasso_σ] obtain y where
    lasso_ys: "lasso ?Gσ y [] ys" and
    "y ∈ set ys" "ys ≠ []" by fastforce
  with ys_in_Z have "y ∈ Z" by blast
```

We will consider two cases: one where ys does not intersect with A, and one where it does. In the former case, we have a cycle in a tangle attractor that does not go to its target set. Per Lemma 5.17, the loop must be won by α, completing our proof for this case. In our Isabelle proof, we have to add the facts that the lasso stays in R and thus exists in $\mathcal{G}[\sigma]$ in R. This is because Lemma 5.17 is used in the context of R; it only applies to lassos in $\mathcal{G}[\sigma]$ in the subgame R.

```
  consider (A_notin_ys) "set ys ∩ A = {}"
         | (A_in_ys) "set ys ∩ A ≠ {}" by blast
  thus "player_wins_list α ys" proof cases
    case A_notin_ys
    with <y∈Z> show ?thesis
      using R_game.tangle_attractor_strat[OF fin_T attr]
      using lasso_restr_V[OF lasso_ys _ ys_in_R]
      by (auto simp: R_subgraph_eq)
```

In the latter case, we show that the highest priority in ys is p. We can show this because ys must contain A, and all nodes in A have the highest priority in R since ys stays in R. Therefore, the highest priority in ys is the highest priority in R, which is p.

```
    case A_in_ys
    with ys_in_R <R⊆V> have "pr_list ys = p"
      using R_game.pr_V_in_list
      by (auto simp: A_def p_def Int_absorb1)
```

Furthermore, we show that α wins p, which is true by the definition of α.

```
    moreover have "player_winningP α p"
      by (auto simp: α_def player_wins_pr_def)
```

Together, these facts show that α wins the loop ys.

```
    ultimately show ?thesis by simp
```

This finishes our proof that all plays that stay in Z are won by α in our step of `search`. □

Now, we show that `search_step` finds tangles.

Lemma 5.33. *If our invariant holds for the initial state (R, Y), then our* `search_step` *relation yields a Y' that contains tangles.*

Proof. We follow along with our Isabelle proof. We do an induction using our `search_step_induct` rule. The start of this proof consists of us getting necessary information from the step and some auxiliary lemmas. First, we use `interpret` so we can use lemmas and definitions in the context of the subgame R. We follow this with shorthand for $\mathcal{G}[\sigma]$ and the vertices in it.

```
lemma search_step_tangles_Y:
  "⟦search_step (R,Y) (R',Y'); search_I (R,Y)⟧ ⟹ ∀U ∈ Y'. ∃α tangle α U"
  unfolding search_I_def split
proof (induction rule: search_step_induct)
  case (step R p α A Z σ Ov Y' Y R')
  then interpret R_game:
    paritygame "Restr E R" "V∩R" "V₀∩R" by blast
  let ?Gσ = "induced_subgraph σ"
  let ?Vσ = "induced_subgraph_V σ"
```

Next, we use some auxiliary lemmas to show that the α nodes in the subgame R are the player-nodes in the region R, and that $\mathcal{G}[\sigma]$ in the subgame R is the same as $\mathcal{G}[\sigma]$ in the entire game restricted to R.

```
  have R_game_V_player_eq: "R_game.V_player α = V_player α ∩ R"
    using restr_subgraph_V_player[OF R_game.paritygame_axioms] .
  have R_game_Gσ_eq: "R_game.induced_subgraph σ = Restr ?Gσ R"
    using restr_ind_subgraph[OF R_game.arena_axioms] .
```

Now, we name some facts from the step, so we can access them later using these names.

```
  from step have
    p_def: "p = pr_set R" and
    α_def: "α = player_wins_pr p" and
    A_def: "A = {v ∈ R. pr v = p}" and
    attr: "R_game.tangle_attractor α T A Z σ" and
    Y'_def: "Y' = (if Ov={} then Y ∪ {S. bound_nt_bottom_SCC R Z σ S} else Y)" and
    R_in_V: "R ⊆ V"
    by auto
```

We show that A is a subset of Z, since it is the target of a tangle attractor. We also show that that Z is a subset of R, and therefore A is also a subset of R.

```
  from A_def <R⊆V> have "A ⊆ Z" "Z ⊆ R" "A ⊆ R"
    using R_game.target_in_tangle_attractor[OF fin_T attr]
    using R_game.tangle_attractor_in_V[OF fin_T attr]
    by auto
```

Furthermore, we use Theorem 5.4 to show that σ is a valid strategy for α with its domain being some number of α-nodes in Z, and under which there exists a path to A from all nodes in Z that stays in R.

```
  have σ_strat: "strategy_of (V_player α ∩ Z) σ"
   and σ_dom: "dom σ ⊆ V_player α ∩ Z"
   and σ_path_to_A: "∀x∈Z. ∃y∈A. ∃xs. path (Restr ?Gσ R) x xs y"
    using R_game.player_strat_in_E
    using R_game.tangle_attractor_strat[OF fin_T attr]
    unfolding R_game_V_player_eq R_game_Gσ_eq
    unfolding strategy_of_player_def strategy_of_def
    by force+
```

Finally, we use Lemma 5.32 to show that all games that stay in Z are won by α.

```
  from step have in_Z_won:
    "∀x∈Z. ∀xs ys. lasso (Restr ?Gσ Z) x xs ys ⟶ player_wins_list α ys"
    using search_step_games_in_Z_won by auto
```

We now start our proof proper by taking a U in Y' that we will show is a tangle. We have two possible cases to consider: either U already existed in Y, or it is a nontrivial bottom SCC in the subgraph of σ in R that lies within Z. Since all U in Y are tangles, the former case is trivial.

```
  show ?case proof (rule ballI)
    fix U assume "U ⊆ Y'"
    with Y'_def consider
      (existing_U) "U ∈ Y" | (new_U) "U ∈ {S. bound_nt_bottom_SCC R Z σ S}"
    by (auto split: if_splits)
    thus "∃α tangle α U" proof cases
      case existing_U with tangles_Y show ?thesis by blast
```

In the latter case, we start by using `interpret` to show that the induced subgame of σ within R is a finite graph. We will use this fact to prove various properties later.

```
    next
      case new_U
      interpret σ_graph: finite_graph_V "Restr ?Gσ R" "?Vσ ∩R"
        unfolding induced_subgraph_V_def
        by unfold_locales force+
```

We use lemmas for nontrivial bottom SCCs in the context of this graph to show various properties. We show that U is part of Z, and part of R. Next, we show that U is finite, that $\mathcal{G}[\sigma]$ restricted to U is strongly connected, that U is closed in $\mathcal{G}[\sigma]$ restricted to R, and that all nodes in U have a successor in U in $\mathcal{G}[\sigma]$.

```
      from new_U <Z⊆R> have "U ⊆ Z" "U ⊆ R" by auto
      from new_U have fin_U: "finite U"
        using σ_graph.nt_bottom_SCC_finite by simp
      from new_U have σ_U_connected:
        "strongly_connected (Restr ?Gσ U) U"
        using σ_graph.nt_bottom_SCC_strongly_connected
        using Int_absorb1[OF <U⊆R>] Int_assoc[of _ "R×R" "U×U"]
        by (simp add: Times_Int_Times) fastforce
      from new_U have σ_U_closed: "(Restr ?Gσ R) `` U ⊆ U"
        using σ_graph.nt_bottom_SCC_closed by simp
      from new_U have σ_U_succ_in_U: "∀v∈U. ∃v'∈U. (v,v') ∈ ?Gσ"
        using σ_graph.nt_bottom_SCC_succ_in_SCC by blast
```

We then begin our proof that U is a tangle for α. We first show that it is nonempty, and that it is part of V, which are relatively simple to prove. The former is true because SCCs are not empty, and the latter is true because U is part of R and R is part of V.

```
      show ?thesis
        unfolding tangle_iff
      proof (rule exI[where x=α]; intro conjI)
        from new_U show "U ≠ {}"
          using σ_graph_nt_bottom_SCC_notempty by blast
        from <U⊆R> <R⊆V> show "U ⊆ V" by blast
```

Next, we show that α wins the highest priority in U. We show this by first showing that U intersects with A. This is true because U is not empty, and we can get a path to a node in A from any node in U. Because U is closed, this means this node in A is also part of U, so U intersects with A. Using this fact, we show that the highest priority in U is p, which we know is true because all nodes in A have priority p, and this is the highest priority in R. Since α is the player who wins p, this means that α wins the highest priority in U.

```
        show "player_winningP α (pr_set U)" proof -
          from <U≠{}> <U⊆Z> σ_path_to_A have "U ∩ A ≠ {}"
            using path_closed_dest[OF _ σ_closed_U] by blast
          with <U⊆R> have "pr_set U = p"
            using pr_le_pr_set[OF fin_U] R_game.pr_set_le_pr_set_V[OF _ <U≠{}>]
            by (simp add: A_def p_def Int_absorb1[OF <R⊆V>]) force
          thus ?thesis by (simp add: α_def player_wins_pr_def)
        qed
```

We will have to give a tangle strategy for the tangle. We have this strategy in the form of σ, but it potentially contains moves for α-nodes outside U, which is more than necessary for the tangle strategy. We take the part of σ that lies in U and name it σ' to get our tangle strategy. We also define shorthand for its subgame $\mathcal{G}[\sigma']$. We then show that this strategy is a subset of the original, that $\mathcal{G}[\sigma]$ and $\mathcal{G}[\sigma']$ are equal in U, and that all nodes in U have a successor in U in $\mathcal{G}[\sigma']$.

```
define σ' where "σ' = σ |` U"
let ?Gσ' = "induced_subgraph σ'"
have σ'_le_σ: "σ' ⊆m σ" by (auto simp: σ'_def map_le_def)
from restricted_strat_subgraph_same_in_region[OF σ'_def]
have graphs_equal_in_U: "Restr ?Gσ U = Restr ?Gσ' U" .
with σ_U_succ_in_U have σ'_U_succ_in_U:
  "∀v∈U. ∃v'∈U. (v,v') ∈ ?Gσ'" by blast
```

Now, we show that this strategy is a tangle strategy, first showing that it is a valid strategy for α. This is true because we obtained σ' by restricting an already valid strategy. To deduce this, Isabelle needs the fact that the edges of this restricted strategy are a subset of the edges of the original (for which we have an auxiliary lemma), that the original strategy was valid, and the definitions of σ' and `strategy_of`.

```
show "∃σ. tangle_strat α U σ"
  unfolding tangle_strat_iff Let_def
proof (rule exI[where x=σ']; intro conjI)
  from σ_strat show σ'_strat: "strategy_of (V_player α) σ'"
    using strat_le_E_of_strat[OF σ'_le_σ]
    by (auto simp: σ'_def strategy_of_def)
```

Secondly, we show that the domain of σ' covers all α-nodes in U. We prove this by showing that the domain is a subset of all α-nodes in U, and by showing that all α-nodes in U are a subset of the domain, giving us equality. In the former case, the domain of σ was already a subset of all α-nodes in U, so restricting it to U maintains this property.

```
show σ'_dom: "dom σ' = U ∩ V_player α" proof
  from σ_dom show "dom σ' ⊆ U ∩ V_player α"
    by (auto simp: σ'_def)
```

For the other direction, we fix a node that is a α-node in U, and make a case distinction for whether this node is part of A. In the case that x is a node in A, then it must have a successor in U. Since U is a subset of Z, this means it must have a successor in Z. Now, per Lemma 5.24, x lies in the domain of σ. Since it also lies in U, it also lies in the domain of σ'.

```
next
  show "U ∩ V_player α ⊆ dom σ'" proof (rule subsetI)
    fix x assume x_in_U_α: "x ∈ U ∩ V_player α"
    consider (x_in_A) "x ∈ A" | (x_notin_A) "x ∉ A" by blast
    thus "x ∈ dom σ'" proof cases
      case x_in_A
      from x_in_U_α σ'_U_succ_in_U <U⊆R>
      have "Restr E R `` {x} ∩ U ≠ {}"
        using ind_subgraph by fast
      with x_in_A x_in_U_α <U⊆Z> show ?thesis
        using R_game.tangle_attractor_strat_in_dom_A[OF fin_T attr]
        by (auto simp: R_game.V_player_eq σ'_def)
```

For the case where x does not lie in A, it still lies in Z since $U \subseteq Z$, so it must lie in the domain of σ per Lemma 5.25. By the definition of σ', this means it also lies in the domain of σ'.

```
      next
        case x_notin_A with x_in_U_α <U⊆Z> <U⊆R> show ?thesis
          using R_game.tangle_attractor_strat_in_dom_not_A[OF fin_T attr]
          by (auto simp: R_game_V_player_eq σ'_def)
      qed
    qed
  qed
```

Furthermore, the range of σ' lies in U because every node in U has a successor in U. This includes all nodes in the domain of σ', which will only have one successor, so their only successor lies in U.

```
  from σ'_U_succ_in_U show σ'_ran: "ran σ' ⊆ U"
    using ran_restrictD[of _ σ U] ind_subgraph_to_strategy
    unfolding σ'_def by fastforce
```

The tangle subgraph of U and σ' is also strongly connected. U is strongly connected in $\mathcal{G}[\sigma]$, which means it is also strongly connected in $\mathcal{G}[\sigma]$ restricted to U. Moreover, a tangle subgraph is the same as a restricted induced subgame, so the tangle subgraph is the same as $\mathcal{G}[\sigma']$ restricted to U. Now, since we have shown that $\mathcal{G}[\sigma]$ and $\mathcal{G}[\sigma']$ are the same within U, the tangle subgraph is strongly connected.

```
  from σ_U_connected <U⊆V> σ'_dom σ'_ran
  show "strongly_connected (tangle_subgraph α U σ') U"
    using tangle_subgraph_is_restricted_ind_subgraph
    by (simp add: graphs_equal_in_U)
```

Finally, we show that α wins all cycles in the tangle subgraph of U and σ'. We first show that this tangle subgraph is a subgraph of $\mathcal{G}[\sigma]$ restricted to Z. With `in_Z_won`, we showed that all plays that stay in Z are won by α. Because the tangle subgraph is a subgraph of $\mathcal{G}[\sigma]$ restricted to Z, all cycles in the tangle subgraph are plays in $\mathcal{G}[\sigma]$ that stay in Z, and are thus won by α.

```
  show "∀v∈U. ∀xs. cycle (tangle_subgraph α U σ') v xs
          ⟶ player_wins_list α xs"
  proof -
    from σ'_dom σ'_ran <U⊆Z> <U⊆V> have tangle_subgraph_subset:
      "tangle_subgraph α U σ' ⊆ Restr ?Gσ Z"
      using tangle_subgraph_is_restricted_ind_subgraph
      using restricted_strat_and_dom_subgraph_same_in_region
      by (auto simp: σ'_def)
    from in_Z_won show ?thesis
      using subgraph_path[OF tangle_subgraph_subset] subsetD[OF <U⊆Z>]
      unfolding lasso_def cycle_def by blast
  qed
  ...
```

Together, these properties show that σ' is a tangle strategy for U. Thus, we have now shown that U is nonempty, U lies in V, that the highest priority in U is won by α, and that we have a tangle strategy σ' for U. Therefore, U satisfies all properties of a tangle, completing our proof that U is a tangle. This shows that all regions in Y' at the end of a step of `search` are tangles. □

With these properties proven, we can show that our invariant is preserved at every step, and that the result of `search` gives us a Y such that our invariant holds for the state $(\emptyset, Y)$.

Lemma 5.34. `search_step` *preserves our invariant.*

Proof. Per Lemma 5.30, `search_step` preserves the property that R is a valid subgame. Furthermore, Lemma 5.31 shows that `search_step` preserves the property that Y is finite. Finally, by Lemma 5.33, Y contains tangles. This shows all three properties of our invariant are preserved. □

Our Isabelle proof, which is shown in Listing 5.49, follows the same structure. It uses the lemmas and the definition of `search_I`.

```
lemma search_step_preserves_I:
  "⟦search_step (R,Y) (R',Y'); search_I (R,Y)⟧ ⟹ search_I (R',Y')"
  using search_step_valid_subgame[of R Y R' Y']
  using search_step_finite_Y[of R Y R' Y']
  using search_step_tangles_Y[of R Y R' Y']
  unfolding search_I_def split by fast
```

Listing 5.49: Our steps of search preserve our invariant.

We show the same for the reflexive transitive closure of `search_step`.

Lemma 5.35. *The reflexive transitive closure of* `search_step` *preserves our invariant.*

Proof. The reflexive transitive closure has two possible cases: either the result is obtained with the reflexive step, where no iterations of `search_step` are taken, or the result is obtained through some number of steps. In the former case, clearly the invariant is preserved, as our result is the same as the initial state, for which the invariant holds. In the latter case, per Lemma 5.34 the steps taken preserve the invariant, so the invariant holds for the result. □

Our Isabelle proof (Listing 5.50) performs the same induction on the reflexive transitive closure of `search_step`.

```
lemma search_step_rtranclp_preserves_I:
  "⟦search_step** (R,Y) (R',Y'); search_I (R,Y)⟧ ⟹ search_I (R',Y')"
  apply (induction rule: rtranclp_induct2)
  using search_preserves_I by blast+
```

Listing 5.50: The reflexive transitive closure of our step relation for search preserves the invariant.

With this, we can show that `search` preserves the invariant.

Lemma 5.36. `search` *preserves our invariant* `search_I`*.*

Proof. By the definition of `search` (Listing 5.45), Y must have been obtained in the state $(\emptyset, Y)$ through the reflexive transitive closure of `search_step` from the state $(R, \emptyset)$. Since the invariant holds for the initial state $(R, \emptyset)$, and the reflexive transitive closure of `search_step` preserves the invariant by Lemma 5.35, the invariant must also hold for the resulting state $(\emptyset, Y)$. □

In Isabelle, our proof (Listing 5.51) similarly uses the definition of `search` and Lemma 5.35.

```
lemma search_preserves_I:
  "⟦search R Y; search_I (R,{})⟧ ⟹ search_I ({},Y)"
  using search_step_rtranclp_preserves_I
  by (simp add: search_def)
```

Listing 5.51: The search algorithm preserves our invariant.

Search Always Finds a Tangle

The next property we want to show is that `search` always finds a tangle. We will have to add the caveat that the initial R must be nonempty, since it obviously cannot find tangles in an empty region. As mentioned previously, we do not show this as part of our invariant because it depends on the initial R, which our invariant does not track.

We start the proof by showing that `search` does not find tangles in an empty R.

Lemma 5.37. *If the initial R is empty,* `search` *finds no tangles.*

Proof. If the initial R is empty, then no further steps can be taken using `search_step`. This is true by its definition (Listing 5.44), since it contains the precondition that R is nonempty. Therefore, the only step we can take in the reflexive transitive closure of `search_step` is the reflexive one. By its definition, `search` starts with an empty Y and obtains the resulting Y with the reflexive transitive closure, hence the resulting Y must be the same as the initial, empty Y. Thus, if the initial R is empty, the Y obtained by `search` is also empty. □

In Isabelle, this proof is split into three lemmas. The first (Listing 5.52) states that `search_step` cannot continue from an empty R, which is shown in a short automated proof. This proof uses the `simp` solver and the automatically generated simplification lemmas for `search_step`.

```
lemma search_step_notempty[simp]:
  "¬search_step ({},Y) (R',Y')"
  by (simp add: search_step.simps)
```

Listing 5.52: No further steps of the search algorithm can be taken from an empty region.

The second lemma (Listing 5.53) uses the first, and states that the reflexive transitive closure of `search_step` will yield a Y' that is the same as the initial Y if R is empty. We can use an automated induction proof, but we cannot directly give the empty set as R, as the induction needs a variable. Therefore, we wrap the automated proof with R given as a variable that is equal to the empty set in an Isar proof, so our lemma itself can use the empty set directly.

```
lemma search_step_rtranclp_empty_R:
  "search_step** ({},Y) (R',Y') ⟹ Y' = Y"
proof -
  assume step: "search_step** ({},Y) (R',Y')"
  have "⟦search_step** (R,Y) (R',Y'); R = {}⟧ ⟹ Y' = Y" for R
    by (induction rule: converse_rtranclp_induct2) auto
  from this[OF step] show ?thesis by simp
qed
```

Listing 5.53: The reflexive transitive closure of the step relation can only take the reflexive step from an empty region.

Finally, we can use that lemma to complete the proof (Listing 5.54). The only further information required by the automated prover is the definition of `search`.

```
lemma search_empty_R:
  "search {} Y ⟹ Y = {}"
  using search_step_rtranclp_empty_R
  unfolding search_def by blast
```

Listing 5.54: The search algorithm finds no tangles in an empty region.

Next, we show that the last step in `search` always finds a tangle, which is equivalent to Van Dijk's Lemma 6 [7, Section 4.5].

Lemma 5.38. *The last step of* `search` *always finds a tangle.*

Proof. We will follow our Isabelle proof for this lemma. This states that if we have a step from (R, Y) to (R', Y'), the invariant holds for (R, Y), and R' is empty, then Y' must be nonempty. We use our induction rule `search_step_induct` for our induction proof.

```
lemma search_step_last_Y'_nonempty:
  "⟦search_step (R,Y) (R',Y'); search_I (R,Y); R' = {}⟧ ⟹ Y' ≠ {}"
  unfolding search_I_def split
proof (induction rule: search_step_induct)
  case (step R p α A Z σ Ov Y' Y R')
```

First of all, we get useful properties from our step. We use `interpret` to allow us to use lemmas and definitions in the context of the subgame R. Then, we get the property that Z is an attractor to A for α with the strategy σ in the subgame R, that R is part of V, and that Z and R are equal. The first two of these properties are part of our step, while the latter is true because R' is defined in the step as $R - Z$, and R' is now empty, and since Z is an attractor in R, it has to lie in R.

```
  then interpret R_game:
    paritygame "Restr E R" "V∩R" "V₀∩R" by blast
  from step have attr: "R_game.tangle_attractor α T A Z σ" by simp
  from step have "R ⊆ V" by blast
  from step have "R = Z"
    using R_game.tangle_attractor_in_V[OF fin_T attr] by blast
```

We also define shorthand for $\mathcal{G}[\sigma]$ and the vertices in it, which we will use later.

```
let ?Gσ = "induced_subgraph σ"
let ?Vσ = "induced_subgraph_V σ"
```

Furthermore, we use Theorem 5.4 to show that the strategy σ is a valid strategy for α in R, and that its range lies in R, since it is a tangle attractor strategy. We use these facts to show that $\mathcal{G}[\sigma]$ in R is equal to $\mathcal{G}[\sigma]$ in the whole game restricted to R, which is true for both its edges and nodes.

```
from <R=Z> R_game.tangle_attractor_strat[OF fin_T attr] have
  σ_strat: "R_game.strategy_of_player α σ" and
  σ_ran: "ran σ ⊆ R" by auto
with restr_ind_subgraph_V[OF R_game.arena_axioms _ _ <R⊆V>]
have R_game_Vσ_eq: "R_game.induced_subgraph_V σ = ?Vσ ∩ R"
  using restr_subgraph_strat_of_player[OF R_game.paritygame_axioms]
  by blast
have R_game_Gσ_eq:
  "R_game.induced_subgraph σ = Restr ?Gσ R"
  using restr_ind_subgraph[OF R_game.arena_axioms] .
```

With these facts, we can show that $\mathcal{G}[\sigma]$ in R is a finite graph without dead ends. The fact that it is a finite graph is trivial. The fact that it has no dead ends is true because all nodes in R have a successor, and this is also true in $\mathcal{G}[\sigma]$ in R because σ is valid in R, and because of the former equalities.

```
from σ_strat_R interpret σ_graph_R:
  finite_graph_V_succ "Restr ?Gσ R " "?Vσ ∩ R"
  unfolding induced_subgraph_V_def
  apply (unfold_locales; force?)
  subgoal for v using R_game.ind_subgraph_succ[of v]
    using R_game_Gσ_eq R_game_Vσ_eq induced_subgraph_V_def
    unfolding R_game.strategy_of_player_def R_game.strategy_of_def
    by clarsimp blast
  done
```

Now, we show that there are no open vertices, which is true since all nodes in R must have a successor in R. Consequently, Y' is now the union of Y and the set of all nontrivial bottom SCCs in $\mathcal{G}[\sigma]$ in R that lie in Z.

```
from <R=Z> step have "Ov = {}"
  using R_game.succ by auto
with step have "Y' = Y ∪ {U. bound_nt_bottom_SCC R Z σ U}"
  by presburger
```

We also show that there exists a bottom SCC in $\mathcal{G}[\sigma]$ in R, that is part of Z. We first show that there exist nodes of $\mathcal{G}[\sigma]$ in R, which must be true because σ is a valid strategy for α in R, and Isabelle further requires the equalities we showed earlier. Then, since we have shown that $\mathcal{G}[\sigma]$ in R is a finite graph without dead ends, it must contain a nontrivial bottom SCC, which satisfies our condition.

```
moreover have "∃U. bound_nt_bottom_SCC R Z σ U"
proof -
  from σ_strat_R <R⊆V> <R≠{}> have "?Vσ ∩ R ≠ {}"
    unfolding R_game.strategy_of_player_def
    using R_game.strategy_of_in_E[of _ σ]
    using R_game.ind_subgraph_V_notempty[of σ]
    using R_game.Vσ_eq by blast
  with <R=Z> show ?thesis
    using σ_graph.nt_bottom_SCC_ex
    using σ_graph.nt_bottom_SCC_in_V by blast
qed
```

As Y' consists in part of the set of nontrivial bottom SCCs in $\mathcal{G}[\sigma]$ in R that lie in Z, and there exists at least one such SCC, clearly Y' is nonempty.

```
  ultimately show ?case by simp
qed
```

Since we have already shown that Y' contains tangles in Lemma 5.33, we have thus completed our proof that the last step of `search_step` always finds a tangle. □

Using this lemma, we can show that `search` will find at least one tangle.

Lemma 5.39. *If we have a valid subgame R that is nonempty,* `search` *always finds at least one tangle.*

Proof. Per Lemma 5.38, we know the last step of `search_step` always finds a tangle. Consequently, if the initial R is nonempty, while the resulting R' is empty, then the reflexive transitive closure of `search_step` will also find at least one tangle, since at least one step must have been taken. Hence, as a consequence of its definition, `search` always finds a tangle, since it starts with a nonempty R and ends with an empty R'. □

In Isabelle, we split this proof into two lemmas, shown in Listing 5.55 below. The first extrapolates Lemma 5.38 to the reflexive transitive closure, which is the first part of this proof. It also requires the fact that the invariant is preserved by the reflexive transitive closure of `search_step`. The second lemma completes the proof, showing that `search` itself gives a nonempty Y for a nonempty valid subgame R.

```
lemma search_step_rtranclp_last_Y'_nonempty:
  "⟦search_step** (R,Y) (R',Y'); search_I (R,Y); R ≠ {}; R' = {}⟧ ⟹ Y' ≠ {}"
  apply (induction rule: rtranclp_induct2)
  by (auto simp: search_step_last_Y'_nonempty search_step_rtranclp_preserves_I)

lemma search_nonempty_R:
  "⟦search R Y; valid_subgame R; R ≠ {}⟧ ⇒ Y ≠ {}"
  using search-step_rtranclp_last_Y'_notempy search_I_base
  unfolding search_def by simp
```

Listing 5.55: If we have a valid, nonempty subgame R, the search algorithm finds at least one tangle.

Now we can fully show that `search` finds at least one tangle in any valid, nonempty subgame R, while it finds none in an empty R. The Isabelle proof for this property is shown in Listing 5.56. It is an automated proof using Lemmas 5.39 and 5.37.

Lemma 5.40. *If we have a valid subgame R, then* `search` *will find tangles if and only if R is not empty.*

Proof. Per Lemma 5.39, if R is nonempty then Y is also nonempty. Furthermore, per Lemma 5.37, if R is empty then Y is also empty. Hence, if Y is nonempty then R cannot be empty, since then Y must also be empty. Thus, R is not empty if and only if Y is not empty. □

```
lemma search_Y_notempty_iff_R_notempty:
  "⟦search R Y; valid_subgame R⟧ ⟹ R ≠ {} ⟷ Y ≠ {}"
  using search_empty_R search_nonempty_R by blast
```

Listing 5.56: If we have a valid subgame R, the search algorithm finds tangles if and only if R is nonempty.

The Search Algorithm is Partially Correct

We finally take all our former lemmas to show the partial correctness of `search`. We will show that the resulting Y of `search` is a finite set of tangles that is nonempty if (and only if) R is nonempty.

Theorem 5.5. *If we have a valid subgame R, then* `search` *gives a finite set of tangles Y that contains at least one tangle if R is nonempty.*

Proof. By Lemma 5.29, our invariant holds for the initial state $(R, \emptyset)$. Moreover, it is preserved by `search` per Lemma 5.36, and Lemma 5.28, this shows that the final Y is finite and contains tangles. Finally, Y is nonempty if and only if R is nonempty per Lemma 5.40. □

Our Isabelle proof uses the same lemmas, and is a short automated proof. It is shown below in Listing 5.57.

```
theorem search_correct:
  "⟦valid_subgame R; search R Y⟧
    ⟹ finite Y ∧ (R ≠ {} ⟷ Y ≠ {}) ∧ (∀U ∈ Y. ∃α. tangle α U)"
  using search_I_correct[OF search_preserves_I[OF _ search_I_base]]
  using search_Y_notempty_iff_R_notempty by simp
```

Listing 5.57: The search algorithm is partially correct.

As we have noted before, this does not show all properties that we wish to show. Still missing is the fact that the tangles found by `search` are ones that were not known before. This is a property that we were unable to prove.

5.3.2 Termination Proof

To show that the `search` algorithm terminates, we actually need to show that it terminates from a state from which the invariant holds. This is necessary because `search` does not necessarily terminate if we do not give it a valid region R as input.

To allow us to prove this fact, we define the concept of a predicate relation terminating from a specific state in the theory `PredicateTermination.thy`. We give an inductive definition `trm` (Listing 5.58), which holds for a predicate r and the state a if, for all successors b of a in this relation, r terminates from b.

```
inductive trm for r where
  "∀b. r a b ⟶ trm r b ⟹ trm r a"
```

Listing 5.58: An inductive definition for a predicate terminating from a state.

We show that if a predicate terminates from some state, then there exists a final state. That is, a state obtained from the initial state through the reflexive transitive closure of the predicate relation which is not itself part of the domain of the relation. Our Isabelle proof shows this using an induction with the `trm.induct` rule, properties of the domain of a predicate, and the transitivity of the reflexive transitive closure.

```
lemma trm_final_state:
  "trm r a ⟹ ∃b. r** a b ∧ ¬Domainp r b"
  apply (induction rule: trm.induct)
  apply (simp add: Domainp.simps)
  using rtranclp_trans[of r] by blast
```

Listing 5.59: A predicate that terminates from some state has a final state reachable from that state.

Furthermore, we show that a predicate that is well-founded under an invariant that is preserved by the relation terminates from any state for which the invariant holds. Our Isabelle proof for this property is shown in Listing 5.60. It uses an induction for well-founded predicates and further needs the generated `trm.intros` rules for `trm`.

```
lemma wfP_I_terminates:
  assumes I_init: "I a"
  assumes I_preserved: "⋀a b. I a ⟹ r a b ⟹ I b"
  assumes wfP_under_I: "wfP (λb a. r a b ∧ I a)"
  shows "trm r a"
  using wfP_under_I I_init
  apply (induction rule: wfP_induct_rule)
  by (blast intro: trm.intros I_preserved)
```

Listing 5.60: A predicate terminates if it is well-founded under an invariant.

With this concept defined, we return to `TangleLearning_Search.thy` to show lemmas we will use to prove termination. First, we will need an element that strictly decreases to show well-foundedness.

Lemma 5.41. *At each iteration of* `search_step`*, R strictly decreases if the initial R is a valid subgame.*

Proof. At each step, R' is obtained by removing a tangle attractor to a set of all nodes with the highest priority in R from R. Because a step could be taken, R is not empty, as this is one of the step's preconditions. Therefore, there must exist a node or nodes that have the highest priority in R. Consequently, the tangle attractor must also not be empty, hence R' is always a strict subset of R. □

Our Isabelle proof, shown below in Listing 5.61, uses the induction rule `search_step_induct`. It then uses auxiliary lemmas to show the set of nodes with the highest priority in R must be nonempty, and Theorem 5.1 to show that the tangle attractor must therefore also be nonempty.

```
lemma search_step_R_decreasing:
  "⟦search_step (R,Y) (R',Y'); valid_subgame R⟧ ⟹ R' ⊂ R"
  apply (induction rule: search_step_induct)
  subgoal for R
    using pr_set_exists[OF finite_subset[OF _ fin_V, of R]]
    using paritygame.target_in_tangle_attractor[OF _ fin_T]
    by blast
  done
```

Listing 5.61: R strictly decreases at every step of search.

We use this lemma to show that `search_step` is well-founded under our invariant.

Lemma 5.42. *So long as our invariant* `search_I` *holds,* `search_step` *is a well-founded relation.*

Proof. By the definition of our invariant (Listing 5.46), so long as the invariant holds, R is a valid subgame. Therefore, per Lemma 5.41, R is constantly decreasing. Since R is a subset of V, it is also finite, meaning it eventually leads to the lowest element in the form of the empty set. This shows that `search_step` is well-founded so long as the invariant holds on every state. □

Listing 5.62 shows our Isabelle proof for this lemma. It uses the same argumentation of our pen-and-paper proof above, but it needs additional lemmas. We use the standard library lemma `wf_subset` and definitions `inv_image` and `finite_psubset` to construct the argument that, since R is strictly decreasing, and is finite, the relation is well-founded.

```
lemma search_step_wfP_I:
  "wfP (λs' s. search_step s s' ∧ search_I s)"
  unfolding wfP_def
  apply (rule wf_subset[of "inv_image (finite_psubset) (λ(R,Y). R)"]
  using search_step_R_decreasing by (auto simp: search_I_def finite_subset)
```

Listing 5.62: Our step relation for search is well-founded under our invariant.

This lemma is used to show that `search_step` terminates from our initial state.

Theorem 5.6. *If the initial R is a valid subgame,* `search` *step terminates from the initial state $(R, \emptyset)$.*

Proof. Per Lemma 5.42, our `search_step` relation is well-founded so long as the invariant holds. We know it holds for the initial state by Lemma 5.29, and that it is preserved at each step by Lemma 5.34. Consequently, the relation is well-founded from the initial state, and thus eventually reaches an empty R, at which point it terminates. □

The Isabelle proof (Listing 5.63) uses our definition `trm`. It uses the lemma `wfP_I_terminates` (Listing 5.60) as the basis of its proof. It then remains to be shown that the invariant holds for the initial state $(R, \emptyset)$, that it is preserved at every step, and that the relation is well-founded under the invariant. These are the three lemmas used in our pen-and-paper proof above.

```
theorem search_step_terminates:
  assumes "valid_subgame R"
  shows "trm search_step (R,{})"
  apply (rule wfP_I_terminates[where I=search_I])
  using search_I_base[OF assms]
  using search_step_preserves_I
  using search_step_wfP_I
  unfolding search_I_def split by blast+
```

Listing 5.63: If the initial R is a valid subgame, our step relation terminates from $(R, \emptyset)$.

Having shown that the step relation terminates from an initial state, we want to show that there always exists a result Y for every R in `search`. We need one additional lemma for this proof.

Lemma 5.43. *If our invariant holds for a state (R, Y), but is not part of the domain of* `search_step`*, then R must be empty.*

Proof. For this proof, we will follow our Isabelle proof. This is a proof by contradiction.

```
lemma search_step_final_empty_R:
  assumes I: "search_I (R,Y)"
  assumes final: "¬Domainp search_step (R,Y)"
  shows "R = {}"
proof (rule ccontr)
  assume "R ≠ {}"
```

Because the invariant holds for (R, Y), R is a valid subgame. We use `interpret` so we can use lemmas in the context of R.

```
  from I interpret subgame
    paritygame "Restr E R" "V∩R" "V0∩R"
    unfolding search_I_def split by blast
```

We can now show that, since R is not empty, there exists a step from (R, Y) to some state (R', Y'). This is true because all other preconditions in `search_step` are always satisfiable, with the existence of the tangle attractor being the only one we need to supply as a specific fact to the prover.

```
  from <R≠{}> obtain R' Y' where
    "search_step (R,Y) (R',Y')"
    using subgame.tangle_attractor_exists[OF fin_T]
    using search_step.step[of R _ _ _ _ _ _ _ Y]
    by clarsimp fastforce
```

As a result, the state (R, Y) lies in the domain of `search_step`, but this contradicts our assumption that (R, Y) does not lie in its domain.

```
  hence "Domainp search_step (R,Y)" by blast
  with final show False by blast
qed
```

This completes our proof that any final state must have an empty R. □

With this lemma proven, we can show the existence of a resulting Y for every valid subgame R.

Theorem 5.7. *For every valid subgame R, there exists a resulting Y obtained through* `search`.

Proof. By the definition of `search` (Listing 5.45), we need to find a Y that results from a final state $(\emptyset, Y)$ of the reflexive transitive closure of `search_step` with the initial state $(R, \emptyset)$. Per Theorem 5.6, there should exist a final state (R', Y') from $(R, \emptyset)$. Furthermore, the invariant holds for the initial state by Lemma 5.29, and is preserved by Lemma 5.34, so the invariant holds for (R', Y'). With Lemma 5.43, we now know that this final state has $R' = \emptyset$. Thus, we have a final state that meets the definition of `search`, which means there exists a Y resulting from `search` for every valid subgame R. □

In Isabelle, our proof (Listing 5.64) uses the same lemmas, as well as our lemma for `trm` that states a final state exists for every predicate that terminates.

```
theorem search_result_exists:
  assumes "valid_subgame R"
  shows "∃Y. search R Y"
  using search_I_base[OF assms]
  using trm_final_state[OF search_step_terminates[OF assms]]
  using search_step_final_empty_R[OF search-step_rtranclp_preserves_I]
  unfolding search_def by fast
```

Listing 5.64: For every valid subgame R, search has a resulting Y.

With this, we conclude our termination proof for the `search` algorithm. We have shown that the inner loop always terminates if the invariant holds for the initial state, and consequently that we always have a result for any valid subgame R that is used as input for the algorithm.

5.4 The Solve Algorithm

The formalisation of the `solve` algorithm has not been completed at present, and no properties of the algorithm have been proven. Much like the `search` algorithm, we can see that the algorithm is a while-loop that terminates when the remaining region R is empty. The formalisation will therefore be similar to that of `search`, with an inductive predicate specifying the step relation of the while-loop, and a definition that uses this step relation to get a final state.

Our formalisation includes an attempt at defining `solve`. It was written early in the research and not further developed, meaning it is likely to be incorrect or insufficient. For this reason, we will not present it in this thesis. However, it can be found in the theory `TangleLearning_Solve.thy` in the available artefact for the sake of completeness.

Chapter 6

Related Work

In this chapter, we discuss two pieces of research are directly related to ours. First is a formalisation of parity games and proof of positional determinancy, which we cover in Section 6.1. The other is a termination proof for Zielonka's recursive algorithm for solving parity games, discussed in section 6.2. We will look at these works and discuss where they show similarities to our formalisation, and where they differ.

6.1 Formalisation of Parity Games and Positional Determinancy

Dittmann [8] submitted a proof for the positional determinancy of parity games to the archive of formal proofs. The formalisation of parity games used in the proof differs from ours, and serves as an interesting comparison. The proof itself is also different from ours, though it uses similar principles.

The first difference worth noting is that Dittmann's assumptions for parity games differ from ours. This formalisation allows parity games to be played on infinite graphs with dead ends, whereas ours is restricted to finite graphs without dead ends. Dittmann also determines the winner of a play using the minimum priority encountered infinitely often, while we use the maximum priority encountered infinitely often instead. These differences do not influence our results due to some futher restrictions imposed by Dittmann. The first of these is that the priorities are finite, and the second is that plays that hit a dead end are won by the opponent of the player who owns the final node. This means there will still be a fixed lowest priority that is encountered infinitely often, and a fixed winner is found for a dead end.

The second difference is the use of codatatypes, which we specifically avoided. Dittmann represents plays with lazy lists. These are a codatatype that represent a list that may either continue indefinitely, making it infinite, or terminate early and be finite. Consequently, this formalisation makes use of corecursive functions, coinductive definitions, and coinduction in proofs. Our formalisation avoided this by reasoning about lassos instead.

```
coinductive valid_path :: "'a Path ⇒ bool" where
  valid_path_base: "valid_path LNil"
| valid_path_base': "v ∈ V ⟹ valid_path (LCons v LNil)"
| valid_path_cons: "⟦ v ∈ V; w ∈ V; v→w; valid_path Ps; ¬lnull Ps; lhd Ps = w ⟧
    ⟹ valid_path (LCons v Ps)"
```

Listing 6.1: Dittmann's definition of a valid path [8].

An example of one of these coinductive definitions is Dittmann's `valid_path` definition (Listing 6.1), which is roughly equivalent to our own `path` definition. Its input is a path, in the form of a lazy list. Rather than specifying the starting point and endpoint like in our definition, these are both included in the lazy list. It then says that an empty path, `LNil` is valid, that a path consisting of a single node in V is valid, and that a path with head v in V with tail Ps is valid if Ps starts with a w that is a successor of v, and Ps is a valid path.

Dittmann's definition of an attractor set also takes a different approach from our own. This is defined as a least fixed point of a step relation (Listing 6.2). First defined is the definition of all nodes

that are directly attracted to some set, being the α-nodes that have a successor in the set, and all $\overline{\alpha}$-nodes that only have successors in the set, none of which may be dead ends. This is then used to define a step relation for extending attractors, `attractor_step`, which is similar to taking all three inductive steps we described for the attractor at once. It takes the union of the target set W, the current attractor S, and all nodes directly attracted to it. Finally, the least fixed point of this relation, starting with W is used to get the final attractor. The least fixed point in this case is the point where the relation stops adding to the attractor set, having found the maximal attractor.

```
definition directly_attracted :: "Player ⇒ 'a set ⇒ 'a set" where
  "directly_attracted p S ≡ {v ∈ V - S. ¬deadend v ∧
      (v ∈ VV p    ⟶ (∃w. v→w ∧ w ∈ S))
    ∧ (v ∈ VV p** ⟶ (∀w. v→w ⟶ w ∈ S))}"

abbreviation "attractor_step p W S ≡ W ∪ S ∪ directly_attracted p S"

definition attractor :: "Player ⇒ 'a set ⇒ 'a set" where
  "attractor p W = lfp (attractor_step p W)"
```

Listing 6.2: Dittmann's definition of an attractor set [8].

This approach is interesting because it is more complicated than an inductive set while achieving the same result. Instead, Dittmann justifies the decision by saying it allows the definition of a different induction rule that, according to Dittmann, is more powerful. This rule is shown in Listing 6.3. It assumes we have a region W in V, after which two cases must be shown. The first is that a property P is preserved by the attractor step for player p from W for all S in V that exhibit P. The second is that this P is also true for the union of all S in V that exhibit P. If both cases can be proven, then the property P also true for the attractor to W for player p.

```
lemma attractor_set_induction [consumes 1, case_names step union]:
  assumes "W ⊆ V"
    and step: "⋀S. S ⊆ V ⟹ P S ⟹ P (attractor_step p W S)"
    and union: "⋀M. ∀S ∈ M. S ⊆ V ∧ P S ⟹ P (⋃M)"
  shows "P (attractor p W)"
```

Listing 6.3: Dittmann's induction rule for the attractor set [8].

The induction rule is used twice: once to show that each attractor set has a witness strategy, and once to show that the least fixed point definition of the attractor set is equivalent to a definition as an inductive set. While the former proof is shorter than ours, its approach seems less intuitive and does not follow any established proof.

To prove positional determinancy, Dittmann uses a proof based on Zielonka's proof [28], much like ours, but it follows a different approach from our own. The proof is an induction on the size of the set of priorities in the graph, which is roughly equivalent to our induction on the size of the graph itself. However, we directly try to show that a parity game can be partitioned into two winning regions, and use this to show that each node is won by one of the two players (see Section 4.4); Dittmann's proof (Listing 6.4) instead shows that each node belongs to a winning region, and uses that to show that parity games can be partitioned into two winning regions.

```
theorem positional_strategy_exists:
  assumes: "v0 ∈ V"
  shows "∃p. v0 ∈ winning_region p"

theorem partition_into_winning_regions:
  shows "V = winning_region Even ∪ winning_region Odd"
    and "winning_region Even ∩ winning_region Odd"
```

Listing 6.4: Dittmann's theorems for positional determinancy [8].

This difference in approaches is in part caused by the differences between our definitions. In Dittmann's formalisation, a winning region for a player is the set of all nodes in V for which they have a winning strategy, which means it is already maximal. Our definition is instead equivalent to a

dominion, being any set of nodes in V for which the player has a winning strategy, and this set may not be maximal.

Overall, Dittmann's formalisation is more polished than ours. Its proofs are more concisely written and it makes extensive use of custom notation to shorten definitions. However, it uses some advanced techniques that are not well-documented. These can be harder to understand for those less experienced with Isabelle. Its custom notation may also be confusing to a reader, as symbols communicate their meaning less clearly than definitions written in words. Some of its techniques could be used to make our proofs more concise. Furthermore, its use of infinite paths for plays, rather than our use of lassos, makes it follow the definitions of parity games more closely. This may increase confidence in its validity compared to the validity of our own formalisation.

6.2 Termination Proof of Zielonka's Recursive Algorithm

In a bachelor's thesis, Abraham [1] uses the formalisations by Dittmann [8] as the basis for a termination proof of (a version of) Zielonka's algorithm [28]. This proof is primarily interesting because of its different approach to defining the algorithm, with its definition being more 'concrete' than ours.

Zielonka's original algorithm is an inductive proof for positional determinancy. The version of the algorithm used by Abraham adapts the structure of this proof into a recursive algorithm. Much like the proof, it starts by finding the player α who wins the (in this case) lowest priority in the game. Next, it finds a tentative winning region A by attracting to all nodes of the lowest priority, removing the attracted region from the game, and repeating this step so long as the lowest remaining priority is still winning for α, taking the union of all regions attracted in this process to form A. This step is a later optimisation to the algorithm, as the original attracts only to a single, arbitrary node of the lowest priority. After finding A, the algorithm is recursively applied to the subgame $G \setminus A$ to find its winning regions W_α and $W_{\overline{\alpha}}$, akin to using the induction hypothesis in the original proof. The algorithm then takes the attracted region for $\overline{\alpha}$ to their winning region $W_{\overline{\alpha}}$ in $G \setminus A$, calling it $W'_{\overline{\alpha}}$ (named B in our proof for Lemma 4.3), and checks whether this region is larger than the original $W_{\overline{\alpha}}$. If the region remains unchanged, then the algorithm returns $A \cup W_\alpha$ as the winning region for α, and $W_{\overline{\alpha}}$ as the winning region for $\overline{\alpha}$. If the winning region has expanded, then A was not a winning region for α, so a recursive call is made for the subgame $G \setminus W'_{\overline{\alpha}}$ to obtain the winning regions X_α and $X_{\overline{\alpha}}$ in that subgame. It then returns X_α as the winning region for α, and $W'_{\overline{\alpha}} \cup X_{\overline{\alpha}}$ as the winning region for $\overline{\alpha}$.

```
function Zielonka :: "'a ParityGame ⇒ 'a set × 'a set" where
  "Zielonka G = (if (isEmpty G ∨ infinite V⇘G⇙ ∨ ¬ ParityGame G) then ({},{})
  else let p = minP G; A = gen_attr G p; Z = (Zielonka (G -G A)) in
  (if (attr_stable G p** Z)
    then (add_to_result p A Z)
    else let att = ParityGame.attractor G (p**) (getW Z p**) in
      (add_to_result (p**) att (Zielonka (G -G att)))
  ))"
```

Listing 6.5: Abraham's definition of Zielonka's algorithm [1].

Abraham's definition of Zielonka's algorithm (Listing 6.5) follows a similar general structure, taking a parity game as input and checking whether it is empty or infinite, returning empty winning regions if either is the case. It takes the player who wins the lowest priority, naming them p instead. Abraham looks at two versions of the algorithm that finds A, which are abstracted here as `gen_attr`. Z is then used to store the winning regions in $G \setminus A$. `attr_stable` checks whether attracting to the winning region for the opponent of p extends that region. If it does not, `add_to_result` is used to return the union of A and the winning region for p in Z as the winning region for p and the winning region of the opponent in Z as the winning region for the opponent of p. If the region is extended, then it gets the actual attractor to it and names it *att*. It then makes another recursive call for the subgame $G \setminus att$, and returns the winning region for p from that call as the winning region for p, and the union of *att* and the winning regions for the opponent from the result of that call as the winning region for the opponent of p.

Note the use of `function` in Abraham's definition. This is a less automated version of `fun` that does not automatically prove properties of the definition, including termination. The termination

proof itself is separate, but uses a similar approach to our proofs of termination, showing that the recursive calls always apply to a smaller subgame, eventually reaching the empty game.

The main difference between Abraham's definition and our own is that Abraham's takes concrete steps. It explicitly defines the algorithm's steps, using `let` to set local variables and making recursive calls, leading to a concrete result. In contrast, our algorithms are defined as step relations using an inductive predicate. We might say the predicate does not 'perform' the steps of algorithm, but rather checks whether a state can be reached from a prior state if these steps were followed; it checks whether a given result is a valid one. Abraham is able to do this with relative ease because Zielonka's algorithm is a recursive algorithm, which makes implementing it in a functional programming language simpler than the iterative (looping) algorithms of tangle learning. If we were to rewrite `search` and `solve` to be recursive algorithms, our definitions could also define concrete steps to a result, rather than checking the validity of an existing result.

Chapter 7

Discussion

Our aim has been to work towards a formal correctness proof for the tangle learning algorithm for solving parity games, both to increase confidence in its correctness and to learn how such proofs can be done. In this chapter, we discuss how our results contribute to our goal by addressing our research questions from Section 1.1. We also discuss the limitations of our results and reflect upon the design decisions we made in our formalisations.

In Chapter 4, we showed our formalisations and some key proofs for the basic concepts of parity games. Most importantly, we gave a proof for positional determinancy that was repeated for our formalisation in Isabelle. Next, Chapter 5 showed our formalisations of the concepts specific to tangle learning and gave a partial correctness proof for the algorithm. Specifically, we gave a definition of tangles, and an algorithm for computing tangle attractors. We proved that this algorithm gives a valid witness strategy for the tangle-attracted region, and that the algorithm terminates. Furthermore, we gave a definition of the `search` algorithm and partially proved its correctness, showing that it finds tangles and that it finds at least one in every region.

7.1 Challenges

One of our intentions was to identify challenges in the process of proving the correctness of tangle learning. Doing so may present useful lessons for future proof attempts in Isabelle. In this section, we will discuss the broad challenges encountered in proving correctness of an algorithm like tangle learning in Isabelle/HOL, as well as the specific hurdles relating to these challenges.

First of all, the formalisation of an algorithm requires one to define the involved concepts. In this case, it was necessary to define the concepts of parity games. We chose to do this ourselves, while we could have decided to use the existing formalisations by Dittmann [8]. Giving our own definitions has the benefit of ensuring our familiarity with and understanding of these definitions. However, we encountered difficulties in working with some concepts such as coinduction, as noted in Section 4.2. This led to us working around using coinduction by reasoning about lassos instead of infinite plays. Had we chosen to use Dittmann's formalisations instead, we might have reduced our time investment in this stage of the formalisation process. On the other hand, we would have had to familiarise ourselves with Dittmann's definitions, who uses techniques (i.e., coinduction) that were beyond our expertise. This would have been an additional challenge.

More specific difficulties were encountered in this first stage. Namely, we encountered difficulties in proving the existence of a witness strategy for attractor sets, as noted in Section 4.3. The steps of constructing the attractor set can be taken in a nondeterministic order, which posed a challenge in proofs that followed a structural induction. Our ultimate solution uses ranks inspired by Zielonka [28], removing the nondeterminism in the order of steps as each rank is always the same. Furthermore, these ranks are defined as a recursive function, rather than an inductive one, which allows for a simpler induction proof on the number of layers. We also initially faced difficulties in proving positional determinancy of parity games, as we noted in Section 4.4. This was once again solved by using a proof based on Zielonka's [28] approach.

The second broad challenge encountered in formalising a correctness proof is the definition and proving of the algorithm itself in Isabelle/HOL. Specifically, tangle learning is an imperative algorithm with while-loops, which can be difficult to formalise in a functional language as is used in Isabelle/HOL.

Our solution is to define a step relation for each iteration of a while-loop and use an invariant to prove properties of the algorithm. This solution was also required for tangle attractors, which now included their strategy as part of their definition. Alternatively, the algorithm could have been rewritten to be recursive. As noted in Section 6.2, a recursive algorithm is easier to define in a functional language. This could be a solution in cases where the exact definition of the algorithm is less important.

There are three main lessons we can take away from these challenges. We learn that inexperience is a major barrier to completing formal proofs in Isabelle/HOL. For one, we see this in our difficulties with coinduction. This was also a major source of delay in our efforts to complete our proofs. However, we also learn that established informal proofs may offer solutions when initial attempts fail. Finally, the functional programming language used by Isabelle makes proving imperative algorithms more difficult, which we have solved using inductive predicates for step relations.

7.2 Van Dijk's Correctness Lemmas

We aimed to use Van Dijk's ten lemmas for the correctness of tangle learning [7, Section 4.5] as a basis for our own proofs. In Table 7.1, we show for each of Van Dijk's lemmas whether they have an equivalent proof in our formalisation. Some of these lemmas no longer apply one-to-one to the versions of `solve` and `search` used in our formalisation. As a result, lemmas 1 and 3 have parallels that are not exactly the same, but they do show a similar property. Furthermore, Lemma 5 is to be split into multiple lemmas in our formalisation. We have not completed formal proofs for Lemmas 4, 5, 8, 9, and 10. At the current stage of the proof these are not yet required, hence their omission.

#	**Lemma**	**Equivalent**
1	*All regions recorded in r in the* `search` *algorithm are α-maximal in their subgame.*	The updated version of `search` no longer tracks regions, but it does find a region Z with the tangle attractor. This region is α-maximal by definition, as it is tangle-attracted. Our Isabelle proof for this fact is shown in Listing 5.41.
2	*All plays consistent with σ that stay in a region are won by player α.*	We proved this property with Lemma 5.32.
3	*Player $\overline{\alpha}$ can reach a vertex with the highest priority p from every vertex in the region, via a path in the region that is consistent with strategy σ.*	Lemma 5.14, part of the correctness of tangle attractors, is a direct parallel. We show that a path exists from any node in the region to a node in A, which will be a vertex with priority p in the context of `search`.
4	*For each new tangle t, all successors of t are in higher α-regions.*	No equivalent lemma exists in our formalisation.
5	*Every nontrivial bottom SCC B of the reduced region restricted by witness strategy σ is a unique p-tangle.*	With Lemma 5.33, we show that `search` finds tangles. It still remains to be shown that these tangles were previously unknown.
6	*The lowest decomposition in the region always contains a tangle.*	Lemma 5.40 shows that `search` always finds at least one tangle if the initial region is nonempty.
7	*A tangle t is a dominion if and only if $E_T(t) \neq \emptyset$.*	We show this directly with Lemma 5.1.
8	*$E_T(t) = \emptyset$ for every tangle t found in the highest region of player α.*	No equivalent lemma exists in our formalisation.
9	*The* `search` *algorithm terminates by finding a dominion.*	The updated version of `search` no longer returns a dominion when it terminates. Instead, we must show that a dominion is eventually found, but we have no such lemma.
10	*The* `solve` *algorithm solves parity games.*	We have not defined `solve`, hence no equivalent for this lemma exists.

Table 7.1: Van Dijk's correctness lemmas and their equivalents in our formalisation.

7.3 Further Properties

Rather than focusing entirely on Van Dijk's lemmas and proofs as a basis for our formal proof, we also aimed to look for alternative and additional properties required for this proof. Here, we will look at these properties.

We find that our proofs themselves closely follow Van Dijk's proofs, but we must show various additional properties that are necessary to prove Van Dijk's lemmas. Often, these are basic properties of parity games and tangle learning that an informal proof would take as obvious, while Isabelle needs further convincing. For example, we needed to show that (tangle) attractors were entirely contained in the game. This fact is clearly true from their definition, but Isabelle required induction proofs to show it.

Other additional properties are necessary as intermediate steps for correctness. For instance, we see in our invariant for `search_step` in Listing 5.46 that we make explicit that the region R is a valid subgame, which is necessary for any of the other properties to be provable. Furthermore, some of our correctness proofs needed more specificity than originally given by Van Dijk. Namely, our proofs showing that `search` finds at least one tangle needed to further specify that this is true *if and only if* the input region of `search` is nonempty, as seen in Lemma 5.40.

Perhaps the most significant property we prove that is not an explicit part of Van Dijk's lemmas is termination. Van Dijk does prove the worst-case complexity for `search` [7, Section 6], which effectively implies termination as a consequence. However, we do not use the same reasoning in our termination proof for `search`, as the worst-case complexity is more than is necessary for showing termination.

7.4 Validity of Our Definitions

The validity of our definitions is important for our proofs to be accepted. Isabelle proofs effectively give us a 'yes-or-no' answer for whether a property holds for our formalisation. However, this is only true if our definitions are free of errors.

We have two ways of mitigating this issue. The first of these is to stay close to the definitions on paper when we write our formalisations. If a reviewer can clearly recognise the definition in Isabelle/HOL as the same one given on paper, they will have greater confidence in its validity. With some definitions, this was straightforward, but due to our use of locales that fix players, this was not always the case.

Take for example our definition of a tangle, its strategy, and its subgraph. Initially, tangles were defined in an instance of the locale `player_paritygame`. When we define them in the locale `paritygame`, they do not look like their definitions on paper. Furthermore, proofs outside a `player_paritygame` context now have additional hurdles to intuitively proving properties of tangles. Therefore, we defined a series of lemmas (Listing 7.1) that show how our definitions are similar to those on paper, while additionally letting us reason intuitively about their properties in a `paritygame` context.

```
lemma tangle_subgraph_eq:
  "tangle_subgraph α U σ = E ∩ (E_of_strat σ ∪ ((U ∩ V_opponent α) × U)"
  using P0.player_tangle_subgraph_def P1.player_tangle_subgraph_def
  using V₀_opposite_V₁ by (cases α; simp)

lemma tangle_strat_iff:
  "tangle_strat α U σ ⟷
    strategy_of (V_player α) σ ∧ dom σ = U ∩ V_player α ∧ ran σ ⊆ U ∧
    (let E' = tangle_subgraph α U σ in
      strongly_connected E' U ∧
      (∀v∈U. ∀xs. cycle E' v xs ⟶ player_wins_list α xs)
    )"
  using P0.player_tangle_strat_def P1.player_tangle_strat_def
  by (cases α; simp)
```

```
lemma tangle_iff:
  "tangle α U ⟷ U ≠ {} ∧ U ⊆ V ∧ player_winningP α (pr_set U) ∧
    (∃σ. tangle_strat α U σ)"
  using P0.player_tangle_def P1.player_tangle_def
  by (cases α; simp)
```

Listing 7.1: Utility lemmas showing our tangles are based on their definition on paper.

Another means of increasing confidence in the validity of our definitions is to check obvious properties these definitions should exhibit. For example, when we defined a path in the graph, we proved that this definition was equivalent to reachability in the graph; we showed that a node was reachable from another if and only if a path existed from the latter node to the former. Our proof of positional determinancy served a similar purpose. Showing that this property holds with our definitions and constraints increases our confidence that they are valid. Moreover, the lemmas in Listing 7.1 are effectively an instance of this approach as well.

7.5 Design Decisions and Potential Improvements

Our formalisation is in essence a programming project. In any such project, design decisions must be made. In this section, we will reflect on these decisions and discuss their benefits and drawbacks, as well as potential alternatives.

We used locales to build up our definitions. This had the benefit of letting us show properties using the assumptions they required, but no more. For example, basic properties of paths only need a graph to exist, while that graph being finite or not is not relevant. It also lets us simplify some definitions by not requiring certain elements as explicit input, such as the nodes of the player for definitions in `player_paritygame`. However, this decision was not without its drawbacks. Especially the aforementioned locale `player_paritygame` introduced problems. As noted in Section 7.4, intuitive reasoning about definitions in the locale `player_paritygame` was not always possible in a context outside of the locale. This required us to define lemmas to specify their properties intuitively. We can see that this design was a trade-off between convenience for the initial definitions and the ability to intuitively prove their properties outside the context.

Discussed in Section 4.2 is the decision to use lassos to represent plays. This mainly had the benefit of avoiding the use of coinduction, which we found difficult to use. Instead, we could use standard inductions that we were familiar with and able to work with without issues. However, as mentioned in the previous section (Section 7.4), this means our definitions do not fully match the definitions of parity game concepts given in the scientific literature. The two noted alternatives, coinduction and ω-words, might be explored by researchers more experienced with them, or able to familiarise themselves with these concepts.

Another design decision was our use of inductive predicates for the step relations of our algorithms. One consequence of this approach was the explicit need to show termination for these definitions. A further consequence is the fact that this makes our algorithm more 'abstract' in a sense, as noted in Section 6.2. Our algorithm confirms that a state is a successor to another, rather than giving a successor to an input. This has one further drawback: if there are multiple cases in a step relation, they are chosen nondeterministically. Moreover, if there are multiple valid successor states, which one is taken is nondeterministic. We see this in our tangle attractor, where there are three possible cases, and multiple successors might meet the requirements for the `player` case. If we have a state reachable by taking some number of steps with `tangle_attractor_step`, then there might be different combinations of intermediate steps that lead to this result. For the tangle attractor, this means the strategy can be different for different 'paths' from the initial state to the result. However, the attracted region itself should always be the same, since the order of steps should not affect the nodes included in the maximal region. We have not proven this property, as it is not required for our correctness proof.

In Section 6.1, we see least fixed points as an alternative to our use of the reflexive transitive closure of a step relation. This could be used to obtain the final state after a loop or a maximal attracted region. Doing so would make our definitions return a result instead of verifying a given result. However, this solution is no more intuitive than ours and will require extensive rewriting of the definition. Furthermore, it does not remove the requirement of showing termination: we would still have to show that a least fixed point exists for our step function. For instance for the tangle

attractor, where we would have to show that there is a maximal attracted region that is always the same. The fixed point must then be based on the region alone, as the witness strategy will always be nondeterministic due to it arbitrarily selecting valid successors.

Another alternative would have been the use of the Isabelle Refinement Framework [19] with imperative HOL [5]. This would have let us formalise the imperative algorithm as-is, without needing step relations. Furthermore, using this framework would have made it simple to prepare our definitions for future adjustments required to generate a verified implementation of the algorithm. The main reason this approach was not taken is that it would have required us to learn how to work with the framework.

Overall, we see various alternatives to our approach to the formalisation of tangle learning. These may have made our proofs or definitions easier, shorter, or more readable. However, they may have conversely made it more difficult for an inexperienced reviewer to understand our formalisation. In general, more advanced techniques make proofs more difficult to understand. This presents a trade-off between readability and efficiency.

Chapter 8

Conclusion

In our research, we aimed to work towards a correctness proof for tangle learning. With this, we both improve confidence in the correctness of the algorithm, and show how such a proof may be done in Isabelle/HOL. To this end, we sought to answer one primary research question supported by three subquestions:

- *How can the correctness of tangle learning be proven in Isabelle/HOL?*
  1. *What are the challenges in proving the correctness of tangle learning in Isabelle/HOL?*
  2. *How can Van Dijk's lemmas for the correctness of tangle learning be proven in Isabelle/-HOL?*
  3. *Which alternative properties should be proven to aid in a correctness proof in Isabelle/HOL?*

Our first step in the formalisation of tangle learning was the formalisation of parity game concepts, which posed the first broad challenge in the proof. In this stage, we encountered specific difficulties with concepts that were beyond our experience. We also found that in some proofs intuitive steps could not easily be proven. These difficulties were mainly overcome by adhering to established informal proofs as a basis for our formal proofs.

The second step was the formalisation of the tangle learning algorithm and related concepts, such as tangles and tangle attractors, posing the second broad challenge in our proof. We found that tangle attractors required an alternative approach using an inductive predicate of a step relation to allow the inclusion of the witness strategy in its definition. We reuse this approach for the formalisation of the `search` algorithm, which was necessary due to the algorithm being an imperative algorithm with while-loops, which cannot be directly translated into a functional programming language.

During the second step, we incorporate several of Van Dijk's lemmas [7, Section 4.5] in our proofs. Often, they serve as the basis of our formal proofs, similar to the established informal proofs used in the formalisations of parity games. Not all of Van Dijk's lemmas have been translated, as the proof has not reached a stage were all of them are relevant.

Various additional properties of tangle learning and parity games were proven to support our main correctness proofs. Many are auxiliary lemmas that show basic properties that would be taken as straightforward in an informal proof. These have the additional purpose of showing the validity of our definitions. Furthermore, we give explicit termination proofs for tangle attractors and `search`, which were absent from Van Dijk's paper.

Besides proving basic properties of our definition to show their validity, we have written our definitions to closely resemble their informal definitions. Where this was not possible, additional lemmas were used to show that our formal definitions corresponded to their informal definitions. This gives us confidence in the validity of our definitions.

With this thesis, we have presented a formalisation of a correctness proof for tangle learning in Isabelle/HOL. This formalisation demonstrates techniques that may be used in proving such an algorithm, such as the use of inductive predicates of step relations and invariants to show the properties of while-loops in imperative algorithms. Our formalisation of parity games may also serve as a basis for future proofs for the correctness of parity game algorithms. However, the correctness proof for tangle learning was not completed, showing only part of the necessary properties of `search` and no properties of `solve`. The completion of the proof is left for future work.

8.1 Future Work

Based on our results, we have five main recommendations for future work. These are ways that our formalisation can be built upon or improved. One common element in all future work is that those who would undertake it are advised to familiarise themselves with proofs in Isabelle/HOL first. Lacking the necessary experience is likely to hamper progress, as it did in this research.

- **Completing the Correctness Proof** The most obvious subject for future work is completing the correctness proof for tangle learning. We managed to partially complete this proof, meaning the ultimate goal of showing that tangle learning solves parity games remains to be achieved. Future research could build on our formalisation and finish the proof, or use the lessons learned to write a new, improved proof.
- **Proving Equivalence of Lassos and Plays** We mentioned in Section 7.4 that our use of lassos decreases confidence in the validity of our formalisation. To remedy this, future research might define plays as infinite paths and prove in Isabelle that every play has an equivalent lasso. This would show that any of our definitions that reason about lassos are definitely true for plays as well.
- **Proving Correctness of Tangle Learning Variants** Van Dijk gives different variants of tangle learning [7, Section 4.6], one of which is the version we use. Proving the other variants or the original version correct may be of interest.
- **Proving Correctness of Other Parity Game Algorithms** Tangle learning is not the only algorithm for solving parity games. Our definitions for parity games could be used as a basis for formalisations of and correctness proofs for other parity game algorithms. An obvious candidate is Zielonka's recursive algorithm [28]. Our own proof for positional determinancy already follows the outline of the algorithm, which future research may turn into an explicit correctness proof.
- **Generating a Verified Implementation** Using the Isabelle Refinement Framework [19], future research could adapt our formalisations to define a concrete, imperative version of tangle learning and prove its correctness. The framework would then allow for the generation of a verified implementation of the algorithm. This verified implementation could have various uses, and its performance might be compared to that of an existing implementation of the algorithm.

Bibliography

[1] R. Abraham. *A Formal Proof of the Termination of Zielonka's Algorithm for Solving Parity Games*. Bachelor's thesis. University of Twente. June 2019. URL: http://essay.utwente.nl/78694/.

[2] M. Benerecetti, D. Dell'Erba, and F. Mogavero. "Solving Parity Games via Priority Promotion". In: *Computer Aided Verification*. Ed. by S. Chaudhuri and A. Farzan. Cham: Springer International Publishing, 2016, pp. 270–290. ISBN: 978-3-319-41540-6. DOI: 10.1007/978-3-319-41540-6_15.

[3] A. Bergström and T. Weber. "Verified QBF Solving". In: *Archive of Formal Proofs* (Mar. 2024). https://isa-afp.org/entries/QBF_Solver_Verification.html, Formal proof development. ISSN: 2150-914x.

[4] H. Björklund, S. Sandberg, and S. Vorobyov. "Memoryless determinacy of parity and mean payoff games: a simple proof". In: *Theoretical Computer Science* 310.1 (2004), pp. 365–378. ISSN: 0304-3975. DOI: 10.1016/S0304-3975(03)00427-4.

[5] L. Bulwahn et al. "Imperative Functional Programming with Isabelle/HOL". In: *Theorem Proving in Higher Order Logics*. Ed. by O. A. Mohamed, C. Muñoz, and S. Tahar. Berlin, Heidelberg: Springer Berlin Heidelberg, 2008, pp. 134–149. ISBN: 978-3-540-71067-7. DOI: 10.1007/978-3-540-71067-7_14.

[6] C. S. Calude et al. "Deciding Parity Games in Quasipolynomial Time". In: *Proceedings of the 49th Annual ACM SIGACT Symposium on Theory of Computing*. New York, NY, USA: Association for Computing Machinery, 2017, pp. 252–263. ISBN: 9781450345286. DOI: 10.1145/3055399.3055409.

[7] T. van Dijk. "Attracting Tangles to Solve Parity Games". In: *Computer Aided Verification*. Ed. by H. Chockler and G. Weissenbacher. Cham: Springer International Publishing, 2018, pp. 198–215. ISBN: 978-3-319-96142-2. DOI: 10.1007/978-3-319-96142-2_14.

[8] C. Dittmann. "Positional Determinacy of Parity Games". In: *Archive of Formal Proofs* (Nov. 2015). https://isa-afp.org/entries/Parity_Game.html, Formal proof development. ISSN: 2150-914x.

[9] E. A. Emerson and C. S. Jutla. "Tree automata, mu-calculus and determinacy". In: *Proceedings of the 32nd Annual Symposium of Foundations of Computer Science*. IEEE Computer Society, 1991, pp. 368–377. DOI: 10.1109/SFCS.1991.185392.

[10] E. A. Emerson, C. S. Jutla, and A. P. Sistla. "On model checking for the μ-calculus and its fragments". In: *Theoretical Computer Science* 258.1 (2001), pp. 491–522. ISSN: 0304-3975. DOI: 10.1016/S0304-3975(00)00034-7.

[11] E. A. Emerson, C. S. Jutla, and A. P. Sistla. "On model-checking for fragments of μ-calculus". In: *Computer Aided Verification*. Ed. by C. Courcoubetis. Berlin, Heidelberg: Springer Berlin Heidelberg, 1993, pp. 385–396. ISBN: 978-3-540-47787-7. DOI: 10.1007/3-540-56922-7_32.

[12] T. C. Hales. "Formal proof". In: *Notices of the AMS* 55.11 (2008), pp. 1370–1380. ISSN: 0002-9920.

[13] J. Harrison. "Formal proof—theory and practice". In: *Notices of the AMS* 55.11 (2008), pp. 1395–1406. ISSN: 0002-9920.

[14] J. Harrison, J. Urban, and F Wiedijk. "History of Interactive Theorem Proving". In: *Computational Logic*. Ed. by J. H. Siekmann. Vol. 9. Handbook of the History of Logic. North-Holland, 2014, pp. 135–214. DOI: 10.1016/B978-0-444-51624-4.50004-6.

[15] M. IJbema. "Perfect Number Theorem". In: *Archive of Formal Proofs* (Nov. 2009). https://isa-afp.org/entries/Perfect-Number-Thm.html, Formal proof development. ISSN: 2150-914x.

[16] M. Jurdziński. "Deciding the winner in parity games is in UP ∩ co-UP". In: *Information Processing Letters* 68.3 (1998), pp. 119–124. ISSN: 0020-0190. DOI: 10.1016/S0020-0190(98)00150-1.

[17] M. Jurdziński. "Small Progress Measures for Solving Parity Games". In: *STACS 2000*. Ed. by H. Reichel and S. Tison. Berlin, Heidelberg: Springer Berlin Heidelberg, 2000, pp. 290–301. ISBN: 978-3-540-46541-6. DOI: 10.1007/3-540-46541-3_24.

[18] M. Jurdziński and R. Lazić. "Succinct progress measures for solving parity games". In: *2017 32nd Annual ACM/IEEE Symposium on Logic in Computer Science (LICS)*. IEEE, 2017, pp. 1–9. DOI: 10.1109/LICS.2017.8005092.

[19] P. Lammich. "Refinement to imperative HOL". In: *Journal of Automated Reasoning* 62.4 (2019), pp. 481–503. DOI: 10.1007/s10817-017-9437-1.

[20] K. Lehtinen et al. "A Recursive Approach to Solving Parity Games in Quasipolynomial Time". In: *Logical Methods in Computer Science* 18.1 (2022), pp. 8–1. DOI: 10.46298/lmcs-18(1:8)2022.

[21] M. Luttenberger, P. J. Meyer, and S. Sickert. "Practical Synthesis of Reactive Systems from LTL Specifications via Parity Games". In: *Acta Informatica* 57.1-2 (2020), pp. 3–36. DOI: 10.1007/s00236-019-00349-3.

[22] R. McNaughton. "Infinite games played on finite graphs". In: *Annals of Pure and Applied Logic* 65.2 (1993), pp. 149–184. ISSN: 0168-0072. DOI: 10.1016/0168-0072(93)90036-D.

[23] T. Nipkow, M. Wenzel, and L. C. Paulson. *Isabelle/HOL: a proof assistant for higher-order logic*. Berlin, Heidelberg: Springer Berlin Heidelberg, 2002. ISBN: 978-3-540-43376-7.

[24] M. O. Rabin. "Decidability of second-order theories and automata on infinite trees". In: *Transactions of the american Mathematical Society* 141 (1969), pp. 1–35. DOI: 10.1090/S0002-9947-1969-0246760-1.

[25] M. Rau. "Earley Parser". In: *Archive of Formal Proofs* (July 2023). https://isa-afp.org/entries/Earley_Parser.html, Formal proof development. ISSN: 2150-914x.

[26] C. Stirling. "Local model checking games (extended abstract)". In: *CONCUR '95: Concurrency Theory*. Ed. by I. Lee and S. A. Smolka. Berlin, Heidelberg: Springer Berlin Heidelberg, 1995, pp. 1–11. ISBN: 978-3-540-44738-2. DOI: 10.1007/3-540-60218-6_1.

[27] R. Tarjan. "Depth-First Search and Linear Graph Algorithms". In: *SIAM Journal on Computing* 1.2 (1972), pp. 146–160. DOI: 10.1137/0201010.

[28] W. Zielonka. "Infinite games on finitely coloured graphs with applications to automata on infinite trees". In: *Theoretical Computer Science* 200.1-2 (1998), pp. 135–183. ISSN: 0304-3975. DOI: 10.1016/S0304-3975(98)00009-7.