Formalization of Tangle and Tangle Learning Algorithm

BONIFACIUS GERALDO CHRISTIANO, University of Twente, The Netherlands

A parity game is an infinite duration game played on a directed graph by two players with each node labeled with a certain natural number. Both players move a token along the edges of the graph and the winner of a game depends on the parity of the highest labeled number of the nodes occurring infinitely often in a path of the token's movement. Parity game is involved in many formal verification problems such as automaton synthesis and verification, and bounded model checking. Solving a parity game is the process of computing the winner of a parity game; various algorithms have been created for this purpose and one of them, the tangle learning algorithm, is of interest in this research. A proof of the algorithm's correctness written in pen-and-paper has been given and this research defines and proves several underlying concepts involved in the algorithm in Isabelle, a software for interactive theorem proving. Isabelle provides a generic approach in verification of specifications or properties of software and algorithms defined as mathematical theorems. Formal proofs are written in a formal system and mechanically verifiable which makes them more accurate, reliable, and easier to verify. First, we define and prove several underlying theoretical concepts involved in the algorithm, then we give a partial implementation of the algorithm in Isabelle.

Additional Key Words and Phrases: Parity games, Formal proof, Formalization, Isabelle, Tangle learning

1 INTRODUCTION

Parity games are turn-based games played on a finite directed graph [4]. Two players, *Even* and *Odd*, move a single shared *token* along the edges of the graph. Each vertex is labeled with a natural number called the *priority* and owned by (exactly) one of the two players. The token is initially placed on a starting vertex, the owner of the vertex selects a successor vertex for that token, respecting the edges of the graph. This process is repeated by whoever owns the vertex the token lands on, thus forming a (possibly infinite) sequence of vertices (and their priorities) called the *play*. Player Even wins when the highest priority appearing infinitely often in the play is an even number, otherwise Odd wins, hence the name *parity*.

Parity games have many applications such as in model checking of μ-calculus [6], LTL, CTL and controller synthesis [10]. Because parity game is an important tool in many fields, fast and efficient algorithms for solving parity games are very valuable so they are also studied in complexity theory. It has been shown that the problem of solving parity games (i.e. determining the winner) lies in the NP and co-NP complexity class [6], more precisely in the UP and co-UP [8] and also in quasi-polynomial (QP) [3]. It remains an open question whether a polynomial time solution exists.

A parity game algorithm is an algorithm for solving parity games. Solving parity games means deciding which player wins a parity game. Various algorithms have been created for this purpose. One of those algorithms that is of interest in this research is called *tangle learning*, described in a paper by van Dijk [4]. The algorithm is based on the concept of a *tangle*, a strongly connected subgraph of a parity game for which a player has a strategy to win all the cycles in the subgraph. Van Dijk has given an informal proof of the algorithm's correctness and termination in his paper. An *informal proof* here means a proof that is written in natural languages and can not be verified in a mechanical or automated manner.

Isabelle is a software for interactive theorem proving [12] available from `http://isabelle.in.tum.de`. Isabelle allows creation and mechanical verification of a *formal proof* which is a proof written in a formal system [13] and can be verified mechanically according to the system's rules. Isabelle provides several logics for theorem proving, one of them being the Higher Order Logic (HOL) which is the most developed one[1] and we will use in this research (Isabelle/HOL). *Formalization* is the process of creating a formal proof from an informal proof, and the formal proof is called the formalization of the informal proof. A formal proof is more reliable than an informal proof because it is based upon an established system and easier to verify by a peer-reviewer [7].

Contribution

The contribution of this paper is formalization in Isabelle/HOL of:

- definitions of closed subset and dominion, and some lemmas about them,
- definition of tangle and some lemmas about tangle,
- partial implementation of the tangle learning algorithm: the attractor and tangle attractor computation that also computes the attracting strategy.

Along with these, there are also several auxiliary definitions and lemmas, such as reachability and strongly connected component. Also, a small lemma that proves parity games with no dead-ends always have infinite plays. The formalization is available on `https://github.com/CafRdkdD/TangleLearningIsabelle`.

2 PRELIMINARIES

This section explains some background knowledge related to this research. The definitions introduced in this section may also be included in other sections.

2.1 Parity Game

A parity game $\Gamma = (V, E, V_0, \omega)$ is a turn-based infinite game played on a directed finite graph $G = (V,E) : V$ are the vertices, $E \subseteq V \times V$ are the edges, and $\omega : V \mapsto \mathbb{N}$ is the priority function which maps each vertex to a priority number. The priority of a vertex v is denoted by $\omega(v)$, while the highest priority of a set of vertices V is denoted by $\omega(V)$. The winner of a priority is denoted by $w(p)$ where p is the priority.

A parity game is played by two players, *Even* and *Odd*. Each vertex is owned by either Even or Odd. Vertices owned by player Even are denoted by V_0 and player Odd by $V_1 = V \setminus V_0$. Since each vertex is

[1] stated in the official manual "Isabelle's Logics"

TScIT 37, July 8, 2022, Enschede, The Netherlands

owned by either players, i.e. $V = V_0 \cup V_1$ and $V_0 \cap V_1 = \emptyset$, it is enough to only define V_0. An (existence of) edge between two vertices v and w is denoted by $v \rightarrow w$. The successor vertices of a vertex v are denoted by $E(v)$ where $v \rightarrow u$ for all $u \in E(v)$. Throughout the paper, if a player is denoted by α, then the other player is denoted by α^{**}. Van Dijk does not consider finite plays which means vertices with no successors (also called *dead-ends*) are not allowed, i.e. $\forall v \in V. E(v) \neq \emptyset$. We will also not consider dead-ends in this paper.

The game is played by moving a shared token along the edges of the graph. Initially, the token is placed at a starting vertex $v_0 \in V$. If $v_0 \in V_0$ then player Even chooses the successor vertex the token will move to, otherwise Odd chooses. Whichever player owns the vertex the token is at chooses the next vertex will move to. The sequence of vertices the token visited creates a path $P = v_0 v_1 v_2 \ldots$ called the *play* where $\forall n \geq 0 : v_n \rightarrow v_{n+1}$. Since there are no dead-ends, there can only be infinite plays. The vertices occurring infinitely often in a path P is denoted by $inf(P)$. Player Even wins the path P if $\omega(inf(P))$ is an even number, otherwise Odd wins.

A strategy $\sigma : V \mapsto V$ is a (partial) function that maps each vertices to its successor vertices, respecting the edges of the graph. Since strategy in this case is just choosing an edge for each vertex, strategy can be viewed as a subset of the edges of the graph, i.e. $\sigma \subseteq E$. A strategy σ' is called a *strategy for player* α if the domain of σ' is restricted to vertices of player α (V_α), i.e. $\sigma' : V_\alpha \mapsto V$. σ' is a *valid strategy* if it is total, i.e. it assigns a successor for every vertex owned by α. σ' is a *winning strategy* (also must be valid) from vertex $v \in V$ if all plays starting from v that is consistent with σ' is winning for player α. A set of vertices $V' \subseteq V$ is a *winning region* for player α if α has a strategy σ that is winning from all $v0 \in V'$.

A game Γ' with graph G' is called a *subgame* of Γ with graph G (denoted $\Gamma' \subseteq \Gamma$) if G' is a subgraph of G. Given a set of vertices V', we say $\Gamma \setminus V'$ to be the subgame of Γ where G' is a subgraph of G with V' (and the corresponding edges) removed, i.e. $V_{G \setminus V'} = V_G \setminus V'$ and $E_{G \setminus V'} = E_G \setminus (V' \times V')$. Additionally, since an edge in a game Γ does not mean the edge is also in a subgame $\Gamma' \subseteq \Gamma$, we write $v \rightarrow_{\Gamma'} w$ to denote an (existence of) edge between v and w in subgame Γ' when discussing about subgames.

2.2 Closed Subset and Dominion

The concepts of closed subset and dominion are related to tangles. They allow more convenient proving of lemmas about tangles. The definitions are based on the definitions by van Dijk [4] and Jurdziński's [9].

Definition 2.1 (Closed subset). A non-empty set of vertices $V' \subseteq V$ is called a *closed subset* with respect to valid strategy σ (strategy for player α) if: $\forall v \in V$,

$$\exists w : v \rightarrow w \tag{1}$$

$$v \in V_\alpha \implies \sigma(v) \in V' \tag{2}$$

$$v \in V_{\alpha^{**}} \implies E(v) \subseteq V' \tag{3}$$

This means that all moves in V' that are consistent with σ stay in V'. In other words, σ traps α^{**} in V'.

Definition 2.2 (Dominion). A non-empty set of vertices $D \subseteq V$ is called a *dominion* with respect to valid strategy σ (strategy for player α) if:

(1) D is a closed subset with respect to σ (σ traps α^{**} in D),
(2) all plays in D that is consistent with σ is winning for player α.

Informally, a dominion of player α is a set of vertices where player α has a strategy to force all plays in the dominion to stay in the dominion and win.

2.3 Tangle

The definition of tangle is based on van Dijk's paper [4].

Definition 2.3 (Tangle). A non-empty set of vertices $U \subseteq V$ is called a *tangle* of player α if:

(1) player α has a valid strategy σ such that the graph *(V',E')* with $V' = V \cap U$ and $E' = E \cap (\sigma \cup (U_{\alpha^{**}} \times U))$ is strongly connected,
(2) player α wins all plays that stay in (V',E').

Since player α wins all plays in U with strategy σ, the other player α^{**} is forced to escape the tangle to avoid losing.

2.4 Tangle Learning Algorithm

The tangle learning algorithm mainly consists of two subroutines (or methods):

- `solve`: iteratively searches and removes dominion of the game and computes the winning regions and winning strategies for both players,
- `search`: given a game and a set of tangles, returns the updated set of tangles and a dominion

The `search` algorithm uses an extension of *attractor set* [14] called the *tangle attractor* and the algorithm `extract − tangles` to extract tangles in a region A with respect to a strategy σ.

2.4.1 Tangle attractor. Several algorithms that solve parity games use the notion of *attractor set* [14] (or attractor). An α-attractor in game Γ of a set of vertices A is a set of vertices where player α can force α^{**} to play to A. Let us denote this α-attractor by $Attr_\alpha^\Gamma(A)$. It can be computed using inductive definition for sets: adding vertices in A as the *base step*, and in each *induction step*, 1) adding vertices of V_α in Γ that have at least an edge to A and 2) vertices of $V_{\alpha^{**}}$ in Γ that have all edges to A. To formally define this, let us denote D_i as induction at step i, then,

$$D_0 = A$$

$$\begin{aligned} D_{i+1} = D_i \cup \{v \in V_\alpha | \exists w : v \rightarrow w \wedge w \in D_i\} \\ \cup \{v \in V_{\alpha^{**}} | \forall w : v \rightarrow w \implies w \in D_i\} \end{aligned}$$

$$Attr_\alpha^\Gamma(A) = \bigcup_{i=0}^{\infty} D_i$$

The definition completes at the least fixed point, i.e. $D_{i+1} = D_i$, or there are no more vertices left in the game that is not already in the attractor set.

Suppose a tangle of player α denoted by t. The *escapes* from tangle t, denoted by $esc(t)$, are defined as a set of vertices in $V \setminus t$ where there exists at least an edge from vertices of a^{**} in t to those vertices. In set notation, $esc(t) = \{w | (v \rightarrow w) \wedge (v \in t \cap V_{a^{**}}) \wedge (w \in V \setminus t)\}$. Denote T as a set of tangles and $T_\alpha \subseteq T$ as tangles won by player α.

The tangle attractor $TAttr_{\alpha}^{\Gamma,T}(A)$ extends the definition of attractor by adding vertices of tangles in T_a that have all escapes to the attractor set. To formally define this, let us denote D_i as induction at step i, then

$$D_0 = A$$

$$\begin{aligned} D_{i+1} = D_i \cup \{v \in V_\alpha | \exists w : v \rightarrow w \wedge w \in D_i\} \\ \cup \{v \in V_{\alpha^{**}} | \forall w : v \rightarrow w \implies w \in D_i\} \\ \cup \{v \in t | t \in T_\alpha \wedge esc(t) \neq \emptyset \wedge esc(t) \subseteq D_i\} \end{aligned}$$

$$TAttr_{\alpha}^{\Gamma,T}(A) = \bigcup_{i=0}^{\infty} D_i$$

Since α wins inside the tangle, α^{**} is forced to escape the tangle to avoid losing inside the tangle. However, in iteration *i+1*, the only escapes are to D_i in which α can force the token to reach A where α also wins. Hereafter, we say *tangle-attracted* to mean *TAttr* and *attracted* to mean *Attr*.

2.4.2 The search *algorithm.* The search algorithm, given a game and a set of tangles, computes the updated set of tangles and a tangle that is a dominion. The algorithm recursively decomposes the game into sets of vertices called *regions* such that each region is won by α and can not tangle-attract any vertices, i.e., $TAttr_{\alpha}^{\Gamma,T}(A) = A$. Given a game Γ, the algorithm retrieves a set of vertices with the highest priority in the game H, then computes a *region* Z by adding all vertices that are tangle-attracted to H. The next iteration repeats this procedure with the subgame $\Gamma \setminus Z$, which repeats again until there are no more vertices left. Thus, each iteration creates a region where each region contains a highest priority that is unique to other regions since each iteration computes a region with decreasing highest priority. For more details about the algorithm, see van Dijk's paper [4].

Algorithm 1 The search algorithm

```
1:  function search(Γ,T)
2:      while True do
3:          r ← ∅, Y ← ∅
4:          while Γ \ r ≠ ∅ do
5:              Γ' ← Γ \ r; T' ← T ∩ Γ';
6:              p ← ω(V_Γ'); α ← w(p);
7:              H ← {v ∈ V_Γ' | ω(v) = p};
8:              Z, σ ← TAttr_α^{Γ',T'}(H);
9:              A ← extract − tangles(Z, σ);
10:             if ∃t ∈ A : esc(t) = ∅ then
11:                 return T ∪ Y, t;
12:             end if
13:             r ← r ∪ Z; Y ← Y ∪ A;
14:         end while
15:         T ← T ∪ Y;
16:     end while
17: end function
```

2.4.3 The extract − tangles *algorithm.* The algorithm as given in van Dijk's paper [4]. The extract − tangles algorithm searches for tangles won by player α in a region A with respect to strategy σ (σ *is strategy for player* α). First, A is reduced by removing all vertices of player α^{**} that have all escapes in the lower regions (regions with lower highest priority) in the subgame Γ' (see Algorithm 1) and all vertices of player α that are constrained in A by σ. Then, each bottom strongly connected components in the reduced region is a tangle. For more details, see van Dijk's paper.

2.4.4 The solve *algorithm.* The solve algorithm computes the winning regions and winning strategies for both players of a game. It recursively decomposes the game, each time reducing the subgame with the attractor set of the dominion in the subgame found using the search algorithm. In each recursion, the attractor set of the dominion is the winning region of either player. For more details about the algorithm, see van Dijk's paper.

2.4.5 Isabelle keywords and terminologies. Here, we explain some keywords and terminologies in Isabelle included in this paper. The highlighted words are the keywords in Isabelle. The **function** keyword declares a recursive function. To declare a non-recursive function, also called *definition*, **definition** is used. **type_synonym** creates a synonym for (combinations of) already existing types. An **inductive_set** is Isabelle's equivalent to the concept of an inductively defined set. The definition of an inductive set are rules of which the set is inductively defined (base steps and induction steps).

A lemma is declared using the **lemma** keyword. The statement of a lemma can be a chain of propositions with the last proposition being the conclusion (e.g. `A` $\implies$ `B` $\implies$ `C` where `A` and `B` are the assumptions and `C` is the conclusion that needs to be proven), or a combination of **fixes**, **assumes**, and **shows** keywords. Propositions declared after **assumes** are the assumptions, each separated by spaces or the optional **and**. Propositions that we need to prove, i.e. the conclusions are declared after **shows**, also separated by spaces or the optional **and**. **fixes** is optional, it is used to declare parameters that will be used in the assumptions and/or conclusions which is not required by Isabelle. The **unfolding** keyword refines the conclusion by expanding definition(s) in the goal to their definition body. The **using** keyword adds facts that may help in proving a lemma or another fact, and the **by** keyword proves the lemma or fact using a proof method, such as `simp`, `auto`, `blast`, etc.

A **locale** in Isabelle is a class that **fixes** some parameter(s) (or variable(s)) and **assumes** some properties about these parameter(s) without knowing their value. From a logical perspective, locales are just contexts that have been made persistent [2]. Anything (functions, definitions, lemmas, etc) declared inside a locale can refer to the fixed parameter(s) of the locale and use the assumption(s) of the locale as fact(s). For each locale, Isabelle automatically creates a *locale definition* which is a definition with equally qualified name as the locale that has the locale's parameter(s) as the definition's parameter(s), the logical conjunction of all locale's assumption(s) as the definition's body, and returns true if the parameter(s) fulfills all the assumption(s). From outside a locale, any functions and definitions inside the locale is added with the parameter(s) of the locale before the actual parameter(s), and any lemmas are also added with the assumption(s) of the locale before the actual assumptions. The fully

qualified name of a function/definition inside a locale (locale name + function/definition name) is used to refer to that function/definition as seen from outside the locale, i.e. it takes all parameters, including those of the locale's. Similarly, the fully qualified name of a lemma inside a locale refers to that lemma that includes all the assumptions, including those of the locale's. The fully qualified name can be used from both outside and inside the locale.

3 METHODOLOGY

Isabelle/HOL is used as the formal theorem prover [12] in this research. Isabelle is an interactive theorem prover, meaning that there is no need for any "compilation" process to get any results. The software does all checks and proving (type inference, declaration checking, natural deduction, etc) as you type which makes it very convenient and more time-efficient. There are 2 methods of writing proofs in Isabelle, using apply-scripts or using the Isar proving language [11]. We will be using Isar instead of writing apply-scripts as it allows more robust proof structure and improve readability.

Furthermore, the Archive of Formal Proofs (AFP) contains collections of formal mathematical proofs written in Isabelle, including of parity games created by Dittmann [5]. We use this formalization as the basis of our research since it already contains lots of definitions and lemmas about parity games. Some of Dittmann's formalization that are of interest in this research are:

- `ParityGame G` is a locale, the definition checks if `G` is a parity game.
- `ParityGame.VV G p` returns the set of vertices of player `p` in game `G`.
- **`type_synonym`** `'a Strategy = "'a ⇒ 'a"` defines strategy as a function from a vertex to another vertex.
- `ParityGame.strategy G p` σ checks if σ is a valid strategy in game `G` for player `p`, i.e. it assigns a successor for every vertex owned by `p`.
- `ParityGame.winning_strategy G p` σ `v0` checks if σ is a winning strategy from vertex `v0` in game `G` for player `p`.
- `ParityGame.subgame G V'` returns a subgame of `G` with vertices of `G` intersected with `V'`.
- `ParityGame.winning_region G p` returns the winning region of player `p` in `G`.
- `ParityGame.winning_path G p P` checks if `P` is a path in game `G` won by player `p`.
- `Digraph.valid_path G P` checks if `P` is a valid path, a path that respects the edges of the directed graph `G`.
- `Digraph.maximal_path G P` checks if `P` is a maximal path, a path that is either empty or ends on a dead-end, in directed graph `G`. Infinite paths are maximal.
- `vm_path G P v0` is a locale, the definition checks if `P` is a non-empty valid maximal path that starts from vertex `v0` in game `G`.
- `vmc_path G P v0 p` σ is a locale, the definition checks if `P` is a non-empty valid maximal path that starts from vertex `v0` in game `G` and conforms to strategy σ.

There are some drawbacks from using Dittmann's formalization. Dittmann considers dead-ends and infinite graphs in his formalization which makes it quite challenging to work with. Also, Dittmann considers the minimum priority as the winning priority even though many literature consider the maximal priority as the winning priority. This happens because there is no general consensus on whether the winning priority is the lowest or the highest priority in the game [1]. Either way, we follow the definition of Dittmann that considers the minimum priority as winning, to not cause complication later on.

Van Dijk's paper [4] is used as the main reference in this research as the tangle learning algorithm is relatively novel. This research also formalizes the concepts of closed subset and dominion as auxiliary methods for proving tangles and tangle learning algorithm. The concepts of closed subset and dominion themselves contain several small lemmas that can be proven in Isabelle.

4 RESULTS

This section explains the formalization produced during the research. Some of the important lemmas will have their informal proofs explained first then the formal proofs. Most of the lemmas have too long formal proofs, so the proofs are not shown, only the statement of the lemmas are shown instead.

4.1 Locale PG

Most definitions and lemmas in the formal proof are declared inside the locale `PG`. Locales can extend other locales, such is the case with `PG`. The child (extending) locale inherits the fixed parameter(s) and assumption(s) of the parent (extended) locale and can add its own parameter(s) and definition(s). The locale `PG` extends the locale `ParityGame` defined by Dittmann [5] and adding the assumption of finite graph by just using finite vertices:

```
locale PG = ParityGame +
  assumes finite_V: "finite V"
```

Because the vertices are finite, it follows that the edges are also finite which completes the definition of finite graph:

```
lemma finite_E[simp]: "finite E" using finite_V
    valid_edge_set by (simp add: finite_subset)
```

`ParityGame` also contains several parameters involved in parity games which we will use. `V` is the set of vertices, `E` is the set of edges, `V0` is the set of vertices of player Even, and ω is the priority function. `ParityGame` defines 3 assumptions:

(1) `valid_edge_set`: validity of edges, i.e. `E` $\subseteq$ `V` $\times$ `V`,
(2) `valid_player0_set`: validity of player Even's vertices, i.e. `V0` $\subseteq$ `V`,
(3) `priorities_finite`: the game has finite number of distinct priorities, i.e. `finite` (ω ` ` `V`)

4.2 Utilities and auxiliary lemmas

We also create several definitions and lemmas that act as auxiliary methods to help simplify defining and proving later on. They are all contained in the locale `PG` (see subsection 4.1).

`strategy_to_edge_set` σ converts a strategy σ to an equal set of edges.

`winning_prio V'` returns the winning priority from a set of vertices `V'`, corresponds to $\omega(V')$

`winning_prio_nodes V'` returns a subset of `V'` assigned with the winning priority.

`prio_winner p` returns the winning player of a priority number, corresponds to *w(p)*.

`node_winner v` returns the winning player of a vertex `v`.

`nodes_winner V'` returns the winning player of a set of vertices `V'`.

`subgame_PG` is a lemma that proves that if a game G fulfills the assumptions of `PG`, then all subgames of G also fulfills the assumptions of `PG`. Below is the lemma declaration:

```
lemma subgame_PG: "PG (subgame V')"
```

The declaration uses the locale definition of `PG`. The definition takes the same parameter as locale definition of `ParityGame` which is a parity game. Since `subgame V'` is still a parity game, we pass it to the locale definition to prove that it also fulfills the locale's assumptions.

The formal proof of the lemma requires proving the assumptions of `PG` which is the combination of assumptions of `ParityGame` and finite vertices ("`finite V`") for subgames. The assumptions of `ParityGame` for subgames are already proven by Dittmann. We only need to prove finite vertices for subgames. Since the set of vertices of a subgame $G' \subseteq G$ is just the subset of the set of vertices of game G, then if G has finite set of vertices, G' also has finite set of vertices.

One exception that is not declared in `PG` is a lemma that proves parity games with no dead-ends always have infinite plays. It is declared in locale `vm_path` instead since it involves paths. `vm_path` extends `ParityGame` and fixes a (possibly infinite) list of path named `P`. This lemma is not exactly an auxiliary lemma since it can be interesting and useful on its own. However, putting it in another section is a waste of space, so it is added here.

Lemma 4.1. *If a game G has no-deadends, then all valid maximal paths in G are infinite.*

Proof. Proof by contradiction. If a valid maximal path P in G is finite, then the last vertex on the path is a dead-end by definition of maximal path. However, this contradicts the assumption that G has no dead-ends. So, all valid maximal paths must be infinite. □

```
lemma (in vm_path)
    vm_path_infinite_in_no_deadends_game:
  assumes "∀v ∈ V. ¬ deadend v"
  shows "¬ lfinite P"
```

4.3 Closed subset and dominion

4.3.1 Closed subset. The definition of closed subset is a predicate (Boolean function) that checks if a set of vertices is a closed subset with respect to a certain strategy and that strategy is a strategy for a certain player. It takes 3 parameter(s): 1) `V'`, a set of vertices, 2) σ, the strategy that closes the set of vertices, and 3) `p`, the player that owns σ. Below is the definition of closed subset:

```
definition closed :: "'a set ⇒ Player ⇒ 'a
    Strategy ⇒ bool" where
"closed V' p σ ≡ V' ⊆ V ∧ V' ≠ {} ∧
 ParityGame.strategy (subgame V') p σ ∧
 (∀v ∈ V'.
  (¬ Digraph.deadend (subgame V') v) ∧
  (v ∈ VV p** ⟶ ∀w. (v → w) ⟶ w ∈ V')))"
```

The definition is based on the description and equations in Definition 2.1. Line 3 establishes non-emptiness and that `V'` is a subset of `V`. Line 6 (with the universal quantifier in line 5) establishes the non-existence of dead-ends described in equation 1. Line 4 and 6 (along with the quantifier at line 5) establishes equation 2 which we will prove later on. Line 7 establishes equation 3.

Below we prove 2 lemmas about closed subset:

Lemma 4.2. *Suppose G' is a subgame with no-deadends (with set of vertices V') and σ is a valid strategy for player p in subgame G'. If $v \in V'$ is a vertex of player p, then:*

(1) $\sigma(v) \in V'$,
(2) $v \longrightarrow \sigma(v)$.

Proof. Suppose a vertex $v \in V'$ is owned by player p ($v \in V'_p$). From the assumption, σ is a valid strategy in the subgame G' for player p, which means σ is (minimally) restricted to $\sigma : V'_p \mapsto V'$. Therefore, σ maps v to some vertex in V', i.e. $\sigma(v) \in V'$ which proves the first sub-lemma.

The second sub-lemma is trivial to prove. By definition of strategy, σ respects the edges of the graph. So, there must exists an edge between v and $\sigma(v)$ which proves the second sub-lemma. □

```
lemma strategy_in_subgame_with_no_deadends:
 assumes "ParityGame.strategy (subgame V') p σ"
     "∀v ∈ V'.¬ Digraph.deadend (subgame V') v"
 shows "∀v ∈ V'. v ∈ VV p ⟶ v ∈ V'" and
       "∀v ∈ V'. v ∈ VV p ⟶ v → σ v"
```

Lemma 4.3. *Suppose V' is closed with respect to a valid strategy σ owned by player p and v0 is any vertex in V'. Then all valid maximal path conforming to σ starting from v0 stays in V'.*

Proof. Take a random vertex $v0 \in V'$. By definition of closed subset and lemma 4.2, if *v0* is a vertex of player p, then $\sigma(v0) \in V'$. By definition of closed subset, if *v0* is a vertex of player p^{**}, then all successor vertices of v is also in V'. Player p playing consistently with σ will never let the token leave V' and the opponent, p^{**}, can not leave V' in any way. This means that all valid maximal paths that conform to σ starting in *v0* will never leave V'. □

```
lemma vmc_path_in_closed_subgame_lset:
  assumes "closed V' p σ" "v0 ∈ V'"
          "vmc_path G P v0 p σ"
  shows "lset P ⊆ V'"
```

4.3.2 Dominion. The definition of dominion uses the definition of closed subset defined above. Similar to closed subset, it is also a predicate that checks if a set of vertices is a dominion of a certain player with respect to a certain strategy. It also takes the same parameter as closed subset. Below is the definition of dominion:

```
definition dominion :: "'a set ⇒ Player ⇒ 'a
    Strategy ⇒ bool" where
"dominion D p σ ≡ closed D p σ ∧
        (∀v ∈ D. winning_strategy p σ v)"
```

This definition corresponds to the description in Definition 2.2. Line 3 corresponds the first description in the list. Line 4 uses `ParityGame.winning_strategy` defined by Dittmann. The definition is:

```
definition winning_strategy :: "Player ⇒ 'a
    Strategy ⇒ 'a ⇒ bool" where
"winning_strategy p σ v0 ≡ ∀P. vmc_path G P v0 p
     σ ⟶ winning_path p P"
```

`winning_strategy` states that "all valid maximal paths (or plays) conforming to σ that start from `v0` are winning for player `p`". Applying this for all vertices in `D` in definition of `dominion` and the fact that dominion is closed, we get the fact that "all plays conforming to σ that start from some `v` ∈ `D` are winning for player `p` and stays in `D`".

Below we prove 2 lemmas about dominion:

We prove that dominion is a closed subset. Since dominion is a closed subset by definition, it is a trivial proof.

```
lemma dominion_is_closed[simp]:"dominion D p σ
    ⟹ closed D p σ"
  unfolding dominion_def by auto
```

Lemma 4.4. *Suppose D is a dominion of player p with respect to a valid strategy σ. D is a subset of the winning region of player of p.*

Proof. By the definition of subset, we need to prove that for all $v \in D$, then also v in the winning region of p. Take any vertex $v \in D$. Then by definition of dominion, σ is a winning strategy from v for player p. A logically equivalent alternate definition of winning region is if w is in the winning region of p, then there exists a winning strategy from w for player p. Using this alternate definition, v must also be in the winning region of p since σ is the winning strategy from v for player p. Generalizing this to all vertices in D and using the definition of subset, we see that D is a subset of winning region of p. □

```
lemma dominion_subset_of_winning_region:
  assumes "dominion D p σ" "strategy p σ"
  shows "D ⊆ winning_region p"
```

4.4 Tangle

4.4.1 Reachability. Since the definition of tangle includes the notion of *strongly connected*, we need to define what it means for a set of vertices to be strongly connected. But before we can define it, we need to define reachability, a vital concept in defining strongly connected. A vertex w is said to be *reachable* by a vertex v if there exists a path from v to w. A vertex can always reach itself using the trivial empty path. Below, we define reachability in Isabelle using an inductive set:

```
inductive_set reachable_by :: "'a ⇒ 'a set" for
     u where
refl: "u ∈ V ⟹ u ∈ reachable_by u" |
trans: "v ∈ reachable_by u ⟹ v ⟶ w ⟹ w ∈
    reachable_by u"
```

The definition takes a vertex `u` and returns a set of vertices reachable by the vertex. It is defined using 2 rules:

(1) `refl` is the reflective (base) rule which says that the vertex `u` is reachable by itself.
(2) `trans` is the transitive (induction) rule which says that if `v` is reachable by `u` and there exists an edge between `v` and a vertex `w`, then `w` is also reachable by `u`.

We also prove a lemma that relates `valid_path` with our definition of reachability above.

Lemma 4.5. *Suppose $P = v_0 v_1 v_2 \ldots$ is a valid path, $n, n' \in \mathbb{N}$ with $n \leq n'$ and n' strictly less than the length of P ($n < len(P)$). Then $v_{n'}$ is reachable by v_n.*

Proof. Since $n \leq n'$ and $n' < len(P)$, we rewrite $n' = n + j$ with $n \leq (n + j) < len(P)$. Now we prove: if $n + j < len(P)$, v_{n+j} is reachable by v_n using induction on j.

Base step: $j = 0$, prove if $n + 0 < len(P)$, v_n can reach v_{n+0}. Since v_{n+0} is just v_n, by reflection rule that says v_n can reach itself, v_n can reach v_{n+0}.

Induction hypothesis (IH): For some arbitrary but fixed j, if $n+j < len(P)$, v_n can reach v_{n+j}.

Induction step: Assuming IH, prove that if $n + (j + 1) < len(P)$, v_n can reach $v_{n+(j+1)}$. Since P is a valid path, $v_{n+j} \rightarrow v_{n+(j+1)}$. Transitively, since v_n can reach v_{n+j} , $v_{n+j} \rightarrow v_{n+(j+1)}$, and $n + (j + 1) < len(P)$, we prove that v_n can reach $v_{n+(j+1)}$.

Substituting $n + j$ with n', we prove that if $n' < len(P)$, then v'_n is reachable by v_n. □

```
lemma valid_path_reachability:
assumes "valid_path P" "n ≤ n'" "n' < llength P"
shows "(P $ n') ∈ reachable_by (P $ n)"
```

4.4.2 Strongly connected. A set of vertices U is strongly connected if every vertex in U can reach all vertices of U. The definition of strongly connected in Isabelle uses `reachable_by` defined in the previous subsection. Below is the definition:

```
definition scc :: "'a set ⇒ bool" where
"scc V' = (∀v ∈ V'. ∀w ∈ V'. w ∈ reachable_by v
    ) "
```

4.4.3 Strategy defined subgame. In the definition of tangle, we need to check whether the graph *(V',E')* with $V' = V \cap U$ and $E' = E \cap (\sigma \cup (U_{\alpha^{**}} \times U))$ is strongly connected. Since a strongly connected component in a graph is also affected by edges of the graph, we can not use `ParityGame.subgame`. Two graphs with same sets of vertices but different sets of edges may not have the same strongly connected component(s). `ParityGame.subgame` only allows us to intersect the vertices and not the edges. For this purpose, we introduce the definition a strategy defined subgame:

```
definition subgame_strategy :: "'a set ⇒ Player
     ⇒ 'a Strategy ⇒ ('a,'b) ParityGame_scheme
    " where
"subgame_strategy V' p σ =
    G(verts := V ∩ V',
    arcs := (E ∩ (V' × V')) -
        {(u, v)|u v. u ∈ VV p ∧ σ u ≠ v},
    player0 := V0 ∩ V')"
```

Similar to `ParityGame.subgame`, but edges with the head owned by player `p` that is not in σ are removed.

4.4.4 Tangle. The definition of tangle in Isabelle is given below:

```
definition tangle :: "'a set ⇒ Player ⇒ 'a
    Strategy ⇒ bool" where
"tangle V' p σ ≡ V' ⊆ V ∧ V' ≠ ∅ ∧
    ParityGame.strategy (subgame V') p σ ∧
    PG.scc (subgame_strategy V' p σ) V' ∧
    (∀v ∈ V'.
        (¬ Digraph.deadend (subgame V') v) ∧
        (∀P. vmc_path G P v p σ ∧ lset P ⊆ V'
            ⟶ winning_path p P))"
```

Line 3 establishes non-emptiness and that V' is a subset of `V`. Line 4 and 7 (with universal quantifier in line 6) serves the same purpose as the one in definition of closed subset. Line 5 establishes strong connectivity of subgame limited by strategy σ. In line 8 and 9, the `vmc_path` which checks if `P` conforms to strategy σ, `lset P ⊆ V'` which checks if `P` stays in V', and `winning_path` which checks if `P` is winning for player `p`, together with the universal quantifier ∀`P` states that "all paths consistent with σ that stay in V' are winning for `p`" which completes the second description in Definition 2.3.

Below, we prove several lemmas related to tangles.

Lemma 4.6. *Suppose T is a tangle won by player α and α^{**} can not leave T, i.e. α^{**} have no escapes from T. Then, T is a dominion won by α.*

Proof. In any tangle, the losing player can escape from the tangle through the escapes from the tangle (t is a tangle, *esc(t)* are the escapes). However, since the losing player a^{**} have no escapes from T and α win all plays that stays in T, all plays starting from some vertex $v \in T$ is always won by α, which is just the definition of dominion. So, tangle T won by α is a dominion won by α. □

```
lemma unescapable_tangle_is_dominion:
  assumes "tangle T p σ" "∀e ∈ T. e ∈ VV p** ⟶
      (∀f. (e ⟶ f) ⇒ f ∈ T)"
  shows "dominion T p σ"
```

Lemma 4.7. *If D is a dominion won by player α with some strategy σ, then D contains at least a tangle also won by player α with σ.*

Proof. Since D is closed, any plays in D consistent with σ must eventually form a cycle with the vertices in the cycle being a strongly connected component (SCC) in D. Also, since α won all plays in D, this SCC must also be won by α with σ and so, the SCC is a tangle won by player α with σ. □

To formalize the proof of this lemma, we created 2 lemmas: `SCC_in_closed_subset` proves a closed subset contains an SCC, and `dominion_contains_a_tangle` uses the previous lemma to prove dominion contains at least a tangle. Unfortunately, we were **unable to completely prove** the former which means we also can not prove the latter.

```
lemma SCC_in_closed_subset:
  assumes "closed V' p σ"
  shows "∃S. S ⊆ V' ∧ PG.scc (subgame V') S"
lemma dominion_contains_a_tangle:
  assumes "dominion D p σ"
  shows "∃T. T ⊆ D ∧ tangle T p σ"
```

4.5 Tangle learning algorithm

4.5.1 Tangle attractor. When doing the attractor set computation for player α, a commonly neglected but important process is also computing the strategy that attracts the vertices of player α. In Dittmann's formalization, the attractor set computation does not consider computing the strategy when computing the attractor set. The tangle learning algorithm utilises this strategy in its algorithm making the strategy computation important to include. First, we introduce 2 `type_synonym`-s:

- `'a Tangle = "'a set × 'a Strategy"` represents a tangle (`'a set`) won by a player α along with the strategy for α that wins the tangle (`'a Strategy`).
- `'a AttrState = "'a set × 'a Edge set"` represents state of the attractor computation where `'a set` is the set of vertices at an induction step and `'a Edge set` is the strategy that attracts the vertices in `'a set`, represented as a set of edges. This representation allows us to combine strategies which would be hard to do otherwise with strategies being functions.

Then, we formalize attractor set (*Attr*) computation that includes computation of the attracting strategy:

```
function attractor_strategy :: "Player ⇒ 'a
    AttrState ⇒ 'a AttrState" where
"attractor_strategy p Z = (let Z' =
    attr_strategy_step p Z in
      (if Z = Z' then Z
      else attractor_strategy p Z'))"
  by auto
```

It recursively compute the attractor until the if condition states that it has reached a fixed point. `attr_strategy_step` defines an induction step in the attractor set computation. Recursive functions in Isabelle must be proven to be able to terminate in all cases; in this case, Isabelle can automatically prove its termination (`by auto`).

Before we can define tangle attractor, we must define *escapes* and *tangles of a player* which are used in the tangle attractor computation.

```
definition escapes :: "Player ⇒ 'a set ⇒ 'a
    set" where
"escapes p T = {v|u v. (u ⟶ v) ∧
    (u ∈ T ∩ VV p**) ∧ (v ∈ V - T)}"
definition player_tangles :: "Player ⇒ 'a
    Tangle set ⇒ 'a Tangle set" where
"player_tangles p T = {(Tp,σ)|Tp σ. (Tp,σ) ∈ T
    ∧ tangle Tp p σ}"
```

Below, we define the tangle attractor (*TAttr*) that includes computation of the attracting strategy:

```
function tangle_attractor :: "Player ⇒ 'a
    AttrState ⇒ 'a Tangle set ⇒ 'a AttrState"
    where
"tangle_attractor p Z T = (let attr_step =
    attr_strategy_step p Z;
    t_attr_step = tangle_attr_step p Z T;
    Z' = (fst attr_step ∪ fst t_attr_step,
      snd attr_step ∪ snd t_attr_step) in
    (if Z = Z' then Z else tangle_attractor p Z'
        T))"
  by auto
```

Similar to the "normal" attractor computation, it also uses recursion. While `attr_strategy_step` defines an induction step of attractor set computation, `tangle_attr_step` defines an induction step of attracting the tangles. It uses the defined `escapes` to only get tangles with escapes only to `Z`, and `player_tangles` to only get the tangles of player `p`.

5 DISCUSSION

In this paper, we formalize underlying and main concepts related to tangle learning algorithm and parity games in general in Isabelle. We used a predefined formalization of parity games to reduce workload and solve a drawback of usimg this formalization, namely infinite graphs by creating the locale PG.

Closed subset and dominion are quite important concepts in parity games. They simplify the problem of finding winning regions of players which is the main objective of parity game solving algorithms. One of the research question is to prove several properties of tangles that are related to these concepts. Since Dittmann's formalization does not include these concepts, we define them in Isabelle. Besides defining them, we also prove several interesting lemmas about them that may help us prove lemmas about tangles. Although initially we define these concepts in Isabelle to help us in proving several properties of tangles, the definitions and especially, the lemmas can be interesting on their own.

Before we define the concept of tangle, we define several concepts involved in the definition of a tangle, such as the notion of strongly connected and strategy defined subgame. We then define the concept of tangle, a core concept in the tangle learning algorithm. We prove the first lemma and tried to prove the second lemma about tangles as described in van Dijk's paper [4]. Unfortunately, we were unable to completely prove the second one (see Lemma 4.7) due to difficulty and lack of time, so the proofs for both `SCC_in_closed_subset` and `dominion_contains_a_tangle` are ended with the **`oops`** keyword. This keyword allows us to end an incomplete proof without throwing any error. In the proof of `SCC_in_closed_subset`, the last thing we manage to do is proving that any paths in a closed subset eventually form a cycle, a path where the first and last vertices are the same. One idea to complete the proof is to show that since the first and last vertices are the same, all vertices in one such cycle can reach each other which makes them a strongly connected component. Then, this lemma can be used to prove `dominion_contains_a_tangle`.

Next, we give a partial implementation of the algorithm by defining the attractor and tangle attractor computation that also computes the attracting strategy. This is required since the tangle learning algorithm uses the attracting strategy unlike the well-known Zielonka's algorithm [14]. Unfortunately, we were unable to progress further due to difficulties in implementing the `extract − tangles` algorithm and lack of time.

Throughout the research, we encounter some difficulties in proving lemmas in Isabelle. Isabelle has a steep learning curve which makes it hard for beginners to use it. While the Isabelle environment itself is interactive and can even search proofs for the user which makes it somewhat easier to use, some facts/lemmas that seem to have obvious proofs at first glance can actually be hard to prove in Isabelle. Trying to prove lemmas in a certain way does not always work out in the end and may require changing the whole approach. One example is `unescapable_tangle_is_dominion` where we have to adapt the definition of `tangle` and create the lemma `vmc_path_in_closed_subgame_lset` just to prove the lemma.

6 CONCLUSION

The tangle learning algorithm is a novel parity game solving algorithm based on the notion of a tangle. Ensuring the algorithm can indeed solve parity games requires proving the algorithm's termination and correctness. For that purpose, an informal proof has been given and the formalization of the proof will allow more accurate, reliable and easier verification of the proof. In this paper, we contribute to the formalization of the algorithm's proof by formalizing the concept of tangles, the core concept in the tangle learning algorithm, along with other related concepts in an interactive theorem prover: Isabelle. Finally, we define the tangle attractor computation, a variant of the attractor set computation that used in the tangle learning algorithm. Further work for the formalization is to define the tangle learning algorithm itself and then start proving lemmas about the algorithm's termination and correctness in Isabelle.

7 ACKNOWLEDGEMENT

I would like to thank Tom van Dijk for his assistance and guidance regarding parity games and Peter Lammich for his technical assistance for Isabelle and both of them for their useful feedback throughout the research.

REFERENCES

[1] Remco Abraham. 2019. *A Formal Proof of the Termination of Zielonka's Algorithm for Solving Parity Games.* B.S. thesis. University of Twente.

[2] Clemens Ballarin. 2003. Locales and locale expressions in Isabelle/Isar. In *International Workshop on Types for Proofs and Programs*. Springer, 34–50.
[3] Cristian S Calude, Sanjay Jain, Bakhadyr Khoussainov, Wei Li, and Frank Stephan. 2020. Deciding parity games in quasi-polynomial time. *SIAM J. Comput.* 0 (2020), STOC17–152.
[4] Tom van Dijk. 2018. Attracting tangles to solve parity games. In *International Conference on Computer Aided Verification*. Springer, 198–215.
[5] Christoph Dittmann. 2015. Positional determinacy of parity games. *Archive of Formal Proofs* (2015).
[6] E Allen Emerson, Charanjit S Jutla, and A Prasad Sistla. 2001. On model checking for the μ-calculus and its fragments. *Theoretical Computer Science* 258, 1-2 (2001), 491–522.
[7] John Harrison. 2008. Formal proof—theory and practice. *Notices of the AMS* 55, 11 (2008), 1395–1406.
[8] Marcin Jurdziński. 1998. Deciding the winner in parity games is in UP∩ co-UP. *Inform. Process. Lett.* 68, 3 (1998), 119–124.
[9] Marcin Jurdziński and Rémi Morvan. 2020. A universal attractor decomposition algorithm for parity games. *arXiv preprint arXiv:2001.04333* (2020).
[10] Michael Luttenberger, Philipp J Meyer, and Salomon Sickert. 2020. Practical synthesis of reactive systems from LTL specifications via parity games. *Acta Informatica* 57, 1 (2020), 3–36.
[11] Makarius Wenzel. 2007. Isabelle/Isar—a generic framework for human-readable proof documents. *From Insight to Proof—Festschrift in Honour of Andrzej Trybulec* 10, 23 (2007), 277–298.
[12] Makarius Wenzel, Lawrence C Paulson, and Tobias Nipkow. 2008. The isabelle framework. In *International Conference on Theorem Proving in Higher Order Logics*. Springer, 33–38.
[13] Wikipedia contributors. 2022. Formal system — Wikipedia, The Free Encyclopedia. `https://en.wikipedia.org/w/index.php?title=Formal_system&oldid=1071134805` [Online; accessed 4-May-2022].
[14] Wieslaw Zielonka. 1998. Infinite games on finitely coloured graphs with applications to automata on infinite trees. *Theoretical Computer Science* 200, 1-2 (1998), 135–183.