UCLouvain

École polytechnique de Louvain

3D Ludii Games
The Case of the Shibumi Set

Author: **Cédric Antoine**
Supervisor: **Éric Piette**
Readers: **Siegfried Nijssen, Benoît Ronval, Achille Morenville**
Academic year 2024–2025
Master [60] in Computer Science

Contents

Acknowledgments

I would like to express my heartfelt gratitude to my family and friends, as well as to everyone who supported and advised me throughout the completion of this thesis.

Additionally, I extend my sincere thanks to Professor Eric Piette for entrusting me with this thesis topic and for his invaluable guidance as my supervisor.

Furthermore, I wish to express my deep gratitude to Professor Siegfried Nijssen, Achille Morenville, and Benoît Ronval for kindly agreeing to serve as members of the jury for this thesis.

I also acknowledge the computational resources provided by the supercomputing facilities of the Université catholique de Louvain (CISM/UCL) and the Consortium des Équipements de Calcul Intensif en Fédération Wallonie-Bruxelles (CÉCI). These resources were funded by the Fonds de la Recherche Scientifique de Belgique (F.R.S.-FNRS) under convention 2.5020.11, as well as by the Walloon Region.

Finally, I am deeply grateful to ChatGPT and the DeepL tool, which were instrumental in the translation and rephrasing of significant portions of my dissertation.

Abstract

Given the General Game Playing (GGP) community's recent interest in modeling, analyzing, and studying 3D board games within the Ludii GGP system, this thesis focuses on two main phases. The first phase investigates the integration of 3D games into Ludii. The second phase explores the experimentation and development of RL techniques and heuristics to enhance AI agents' performance in these games. Initially, this thesis will focus on Shibumi games, which are generic enough to expand to any 3D game. These games offer strategies that are easy to learn but take a lifetime to master.

As for the current literature, Ludii stands as the first General Game Playing system capable of modeling, visualising, and playing any finite extensive-form game through ludemic modeling and a class grammar approach. However, Ludii was developed as part of the Digital Ludeme Project (DLP), which primarily focuses on traditional board games. As a result, the language, state representation, interface, and AI methods used in Ludii's 1300+ games are not fully optimised for modeling 3D games invented in the last 50-70 years, which fall outside the scope of the DLP.

This paper concludes by highlighting certain limitations as well as several promising results of the carried out implementations. In addition, a number of suggestions and ideas for improvements are put forward in order to improve the obtained results and go further. Done implementations are publicly available at: https://github.com/cedantoine/Ludii/tree/dev.

1

Introduction

Humankind has been playing games since the dawn of time. It is thought that one of the first board games to have been devised, Senet, dates back to ancient Egypt around 3100 BC [1]. With the advent of artificial intelligence, games have once again enjoyed a renaissance, becoming the main testbed for major advances in the field [2]. Indeed, their simplicity and popularity, as well as the ability of humans to perceive the problem (game interface) and its solutions (strategies), make them perfect for this purpose.

In this context, General Game Playing (GGP) has taken on crucial role. The General Game Playing challenge [3], in relation to AI, is to develop agents capable of understanding and effectively playing previously unknown games without human intervention, based solely on their game description. It is important to note that General Game Playing is considered a necessary step in the development of Artificial General Intelligence (AGI) [4]. The development of general agents and subsequent AI techniques are fundamental to the development of real-world agents capable of handling new and unpredictable situations.

Several General Game Playing systems (i.e. game modeling software) currently exist for many types of games, ranging from deterministic perfect-information games [5] to video games [6][7]. This work explores the integration of 3D board games in the Ludii general game system [8], focusing on Shibumi games [9]. Indeed, Shibumi's 3D games, which cover a wide range of game categories (capture games, counting games, puzzles, etc.) while using only a limited set of components (balls, grids, and simple rules), seem generic enough to extend to any 3D game and offer strategies that are easy to learn but take a lifetime to master. Finally, given that the GGP framework supports the development of general-purpose techniques, this work takes advantage of the integration of 3D games to explore and adapt these techniques to 3D games, a game type that remains largely under-explored in this domain.

Hence, the subsequent work is divided into four parts. The first part offers a formal introduction to the key aspects of General Game Playing, the Ludii system, Shibumi and other 3D games. The second part covers the integration of the previously mentioned

1

games into Ludii. The third part focuses on the experimentation and modeling of AI on the newly integrated games. Both implementation and experimental results of new agents are presented and compared with existing ones. Finally, the last part summarises the results obtained and outlines potential directions for future work.

2 Background

The first section of this chapter provides a formal introduction to key aspects and the historical background of General Game Playing and General Board Game Language. The second section introduces the Ludii GGP system. The third section discusses the main features of Shibumi and 3D board games. Finally, the fourth and last section formulates the motivations that led to this work.

2.1 Introductory Notions

The aim of this section is to provide a general overview of the field of GGP.

2.1.1 General Game Playing

As previously indicated, in the field of AI, the aim of the GGP [3] is to develop computer players who are able to learn and interpret the rules of previously unknown games. The goal is for them to learn how to play these games well (understanding the objective and developing effective strategies) without human intervention, using only the game descriptions as resource and having little to no reflection or learning time before the game and limited time during gameplay. This step is considered crucial in the development of real-world artificial agents capable of handling new and unpredictable situations [4].

In the field of GGP it is important to distinguish between GGP systems and GGP agents. GGP Systems (e.g GGP-Base [10] or Ludii [8]), provide the environment in which games are played. They include the necessary infrastructure to parse the game descriptions (written mostly in General Game Languages which we will discuss in the next subsection), simulate the game environment, and manage the interactions between the different GGP agents. GGP agents (e.g. Woodstock [11][12], Cadiaplayer [13] or Ary [14]), on the other side, are the AI programs or approaches developed to play games in the GGP system. They are capable of interpreting the rules, formulating strategies and making decisions during game-play.

Historically, the first GGP model was defined in 1968 by Jacques Pitrat [15] to describe two-player games with full information on rectangular boards. Subsequently, it was only in the 90s that other GGP systems appeared, such as SAL [16], Metagamer [17] and Hoyle [18]. Finally, after the end of the millennium, more and more GGP systems were developed,

such as Multigame [19] and Zillions of Games [20] in 2001 and 2002 respectively. However, the modern era of GGP truly began with the Stanford Logic Group, whose work led to the development of GGP-Base and the Game Description Language (GDL) [5], which will be discussed in the following section.

2.1.2 General board Game Languages

Most GGP systems use a standardised game description language. Approaches to knowledge representation, resonance and learning can vary greatly (e.g. for performance reasons) depending on the format and level of abstraction of the game description. In this subsection, we will introduce GDL, one of the most widely used frameworks, and RBG, known for its efficiency. In the next section we will then discuss the Ludii system, on which this work is based.

Since 2005, Stanford's *Game Description Language* (GDL) and previously mentioned GGP-Base system have become one of the main standards in GGP academic research. GDL is a set of first-order logic clauses, describing games in simple instructions. It has been designed for full information games. However, two extensions, GDL-II [21] and GDL-III [22] have been developed for games containing hidden information and epistemic games respectively. The generality of GDL [23] provides a high-level algorithmic challenge that has led to significant contributions [24] in recent years, particularly in the Monte Carlo tree search improvements [25] [26][27].

Despite its generality, GDL is unfortunately known to be inefficient in certain scenarios [8] [28]. Its logic-based descriptions are not only verbose but also computationally expensive, as each step in the game requires the evaluation of logical rules. This significantly increases the computational load, making it challenging to model or play highly complex games such as Go or Chess. In addition, the lack of modularity in GDL means that even small changes to the rules or structure of a game often require extensive rewriting, limiting its practicality for reuse and adaptation.

In contrast, *Regular Boardgames* (RBG) [29], was developed in 2019 to address these shortcomings by providing a more efficient and flexible dual-language approach. RBG provides both a high-level language for concise, human-readable descriptions and a low-level language optimised for AI reasoning. The high-level version simplifies the task of modifying and understanding game rules, making it more accessible to game designers. Once the high-level description is written, it can be compiled into the low-level version for efficient simulation and AI processing. This duality ensures that RBG strikes a balance between human readability and computational efficiency.

In addition, RBG's streamlined format makes game descriptions shorter and easier to reuse across games, as opposed to GDL's more rigid and verbose structure. An illustrative example of this can be found in Appendix by comparing GDL and RBG implementation of Tic Tac Toe in respectively Figure A.1 and Figure A.2. However, RBG remains restricted to deterministic, perfect-information games, without support for stochastic or hidden-information games.

Both GDL and RBG have been crucial in advancing the understanding and development

of games in AI, providing frameworks that facilitate the exploration of complex game mechanics and strategies. Their contributions have significantly influenced research and practical applications in Machine Learning, Machine Reasoning, and beyond.

2.2 Ludii and The Ludemic approach

As mentioned earlier, Ludii stands out as the first complete GGP system capable of visualising, modelling and playing (by a human or an AI) any finite game of extensive form thanks to ludemic modelling and a class grammar approach [30]. The library of its most recent public version, 1.3.13, released on 10-07-2024, contains 1,500 games of various types, at the time of writing this paper. The structure of its game's library can be seen in the figure 2.1 below (n.b. the wip_cedric folder contains the games resulting from this work).

Figure 2.1: Ludii game library structure

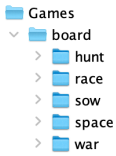

Figure 2.2: Main Ludii game types

2.2.1 The origins of Ludii

Ludii was created as a successor of Ludi, an earlier system developed by Cameron Browne as part of his research into the automated design and evaluation of combinatorial games [31]. Ludi pioneered the field of general-purpose game design with its ability to generate new games by recombining rules from existing games, and to evaluate their quality based on aesthetic and strategic criteria. Ludi's ability to generate playable games [32], such as the acclaimed Yavalath, illustrated AI's potential for game design and analysis.

Nevertheless, Ludi had certain limitations. It was primarily focused on generating new combinatorial games, particularly abstract strategy games, and evaluating their quality through self-play. While it successfully demonstrated the capabilities of AI in game design, its scope was limited to a specific subset of games. Furthermore, the language used in Ludi was not optimal for describing or modelling a wide variety of games, which limited its flexibility. While groundbreaking in its time, Ludi lacked the versatility needed for broader applications, such as the study of traditional games in historical and cultural contexts. One of the limitations that Ludii could now overcome is the creation of games with entirely new mechanisms.

Recognizing all these limitations and the need for a more comprehensive system, Ludii was developed as part of the Digital Ludeme Project (DLP) [33][34]. Using Ludii as a central tool, the DLP aimed to achieve the unprecedented task of cataloging and studying over 1,000 traditional games using AI and data mining techniques [35][36]. Therefore, the goal of Ludii was to go beyond game creation and evaluation, offering a platform for modeling

(while still maintaining its importance, as demonstrated by recent efforts to combine LLM models with Ludii for new game creations [37]), studying, and comparing [38] a wide variety of traditional and modern games. Unlike Ludi, which was more focused on creating new games, Ludii was designed with a broader and more culturally rich goal: to digitally reconstruct and analyze the world's traditional games [39][40].

2.2.2 The Ludemic Approach

At the heart of Ludii's innovative framework is the concept of the ludeme [41], a term first introduced by game historian David Parlett to describe the basic units of game-related information [42]. In simple words, a ludeme can be thought of as the building blocks of a game. These elements represent a wide range of components, including the game's rules, equipment, objectives, player actions and interactions. Just as words form sentences in a language, ludemes are the essential components that together define the structure and play of a game.

Ludemes are more than just pieces or moves. They encapsulate all aspects of a game's design, from the type of board used to the specific victory conditions. They allow games to be expressed in a modular way, where any game can be broken down into a collection of these different parts [43]. This allows for easy creation, manipulation and comparison of games through the reassembly and modification of their core components.

In the ludemic approach, games are described as a combination of these fundamental ludemes, making it easier to analyse and model games across different types, cultures and time periods [44]. In contrast to traditional methods of game representation, such as verbose logical descriptions, ludemes are a more intuitive and flexible way of capturing the essence of a game. Ludii uses ludemes to create a structured representation of games that is both human-readable and computationally manageable.

Each ludeme captures a specific aspect of a game, and ludemes can be grouped together to form more complex structures. For example, the layout of a board, the movement of pieces, or the scoring mechanism can all be expressed as separate ludemes that, when combined, define the complete game.

For example, the classic game of Tic Tac Toe can be described in Ludii by combining a few basic ludemes (see Figure 2.3 below).

```
(game "Tic-Tac-Toe"
  (players 2)
  (equipment {
    (board (square 3))
    (piece "Disc" P1)
    (piece "Cross" P2)
  })
  (rules
    (play (move Add (to (sites Empty))))
    (end (if (is Line 3) (result Mover Win)))
  )
)
```

Figure 2.3: Ludemic implementation of Tic Tac Toe in Ludii
Source: https://github.com/Ludeme/Ludii

The game starts by defining the mode of play with the player ludeme (in this case two players taking turns). The board is then represented as a 3x3 grid defined by the board subset of the equipment ludeme. Another equipment subset assigns each player a specific piece, such as a cross or a disc, which they place on the board in turn. The rules ludeme defines the mechanics of the game, stating that players take turns placing their pieces on empty cells, aiming to align three of their pieces in a row to win. The victory condition is described as another ludeme, where a line of three pieces results in a win for the current player.

By breaking the game down into these modular components, Ludii simplifies the game description into a clear, human-readable format, while maintaining flexibility for customisation. For example, changing the board size from 3x3 to 4x4 or changing the victory condition is as simple as changing a single parameter in the corresponding ludeme, highlighting the adaptability and simplicity of the approach. These advantages can easily be observed by comparing Ludii implementation of Tic Tac Toe in Figure 2.3 to the GDL and RBG ones in Figures A.1 and A.2.

2.2.3 The Ludii system

Ludii's underlying structure is designed using a class grammar approach that links its game description language to the Java classes that represent the game logic. This approach ensures that any game description written in Ludii's language maps directly to the system's Java class structure, creating a direct correspondence between the conceptual representation of games and their computational implementation. Using Java Reflection, Ludii dynamically generates game instances from the game descriptions, which are parsed as trees of ludemes - the basic units of game information.

The Ludeme.java interface, which is at the top of the ludeme class hierarchy, is implemented by each Java class corresponding to a ludeme. They are organised as follows:

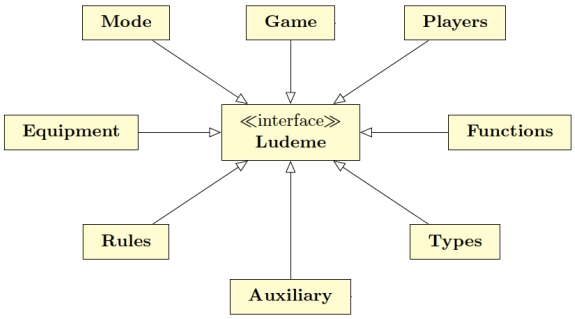

Figure 2.4: Ludemic categorisation in Ludii
Source: [40]

At the root of any game description is the game ludeme, represented by the Game.java class, which encapsulates the primary elements of a game such as its name, players, equipment and rules. Each of these elements is defined as a node in the tree of ludemes,

with specific ludemes corresponding to Java classes. For example, a board is represented by the class Board.java, and players by the class Players.java.

The class grammar approach provides flexibility by representing game elements (ludemes) as independent classes, allowing modular development. Each ludeme is implemented as a separate Java class that can be reused across different games. This modularity makes Ludii highly extensible, as new game elements or features can be easily added without requiring a major rewrite of the system. Furthermore, the class grammar ensures that all games described in Ludii are automatically instantiated into the corresponding Java classes, providing a 1:1 mapping between grammar and code. This reduces redundancy and ensures consistency between game descriptions.

In addition to the class grammar, Ludii uses a tree-based structure for evaluating game states and moves. Each ludeme in the tree corresponds to a Java class, and operations such as evaluating legal moves or determining game results are performed by recursively evaluating the tree. For example, the Context class is responsible for storing the state of the game, the sequence of moves, and any stochastic elements, while interacting with other classes such as State.java and Trial.java to manage the progress of the game. This context-driven design ensures efficient computation of legal moves and state transitions during game-play.

The advantages of Ludii's class grammar approach are numerous, particularly when considering its comparison to other general game systems like Regular Boardgames (RBG) or Game Description Language (GDL) [8][28]:

- **Clarity:** Ludii's use of ludemes provides a clear and concise way to describe game elements, reducing the complexity seen in systems like GDL. By organising the game's components into modular ludemes, Ludii makes game descriptions shorter and more readable, improving the clarity of game logic.
- **Generality:** Ludii is highly general, capable of representing a wide variety of games, from traditional board games to complex 3D games and cultural games. Compared to RBG, which is primarily focused on regular board games, Ludii's generality allows for greater flexibility in handling diverse game types.
- **Extensibility:** The ludemic approach allows Ludii to be easily extended by adding new game components or features with minimal changes to the core system. This modularity provides an advantage over RBG and even more GDL, where incorporating new elements can require more extensive rewriting.
- **Efficiency:** Ludii is highly efficient, particularly in terms of game description size. Games that require many tokens in systems like RBG can be described with far fewer in Ludii. This efficiency not only speeds up the parsing and execution of games but also makes it easier to scale across different types of games.
- **Evolvability:** Ludii is designed to evolve alongside changing game design trends. Its modular structure supports the easy introduction of new game mechanics or systems without having to overhaul existing logic. In comparison, systems like GDL and RBG may require more complex adjustments to adapt to new game features.

- **Cultural Applications** [45]**:** Ludii's ability to model and simulate games from various cultures and historical periods is a distinguishing characteristic. Its framework supports the exploration of how games evolve and spread across different cultures.

2.2.4 AI Agents in Ludii

Ludii, with its expansive and ever-growing library of games, includes implementations of standard game-playing agents mostly focused to be able to operate across a wide variety of games and their variants [46]. These agents primarily leverage general techniques such as Monte Carlo Tree Search (MCTS) [47], one of the most prominent methods in General Game Research. Additionally, Ludii allows third parties to develop custom agents, enabling the integration of tailored strategies or novel AI approaches. The platform also provides visualizations that offer valuable insights into the decision-making processes of algorithms like MCTS, enhancing understanding and facilitating debugging.

In terms of heuristics implemented for the aforementioned agents, Ludii supports methods that prioritize various objectives, such as line creation, piece proximity to the centre or edge of the board, mobility promotion, and more. Previous work has evaluated the performance of the majority of these heuristics on various games and explored predicting a given heuristic's performance based on its ludemic implementation [48]. However, these heuristics have not yet been tested on our games, nor have strategies specifically designed for 3D games been incorporated, such as favouring pieces positioned on higher layers or leveraging three-dimensional spatial arrangements.

2.3 Shibumi and other 3D games

The primary goal of this work is to implement 3D games in the Ludii system. Specifically, we focus on Shibumi games, as their simplicity and versatility make them a suitable starting point for broader 3D game implementations. By adapting Ludii's ludemes to support the Shibumi set, we aim to create a foundation that will allow for the implementation of a wider range of 3D games within the system.

Shibumi is a term from Japanese aesthetics that refers to a particular kind of simple, subtle, and unobtrusive beauty. It implies an elegance achieved through minimalism, where something may appear simple on the surface but reveals hidden complexity and depth the more time is spent with it. In design or art, something described as shibumi is often unassuming and understated, yet possesses an inherent sophistication and richness that comes to light on closer inspection. The concept is often associated with things that are harmonious, balanced and timeless, without the need for excess to convey their beauty. In the context of games, especially abstract ones like those in the Shibumi set, this aesthetic is reflected in the design philosophy: simple rules and components that lead to deep, strategic play.

The games we are going to try to implement are all included in the *Shibumi Rule Book* [9] (see cover in Figure 2.5 bellow). It is interesting to note that two Shibumi games, Spline and Spava, are already implemented in the current version of Ludii (see Figures 2.6 and 2.7). This relies on the fact that even if both of these games use a 3D board, they

solely need 2D features and interactions to be played. Our goal is to build on these initial implementations by fully integrating the 3D mechanics of Shibumi games into the Ludii system, ensuring that both the board and the piece interactions are accurately represented in three dimensions.

Figure 2.5: Shibumi Rule Book
Source: [9]

The Shibumi games are categorised as follows:

- **N-in-a-Row:** Games where the objective is to form lines of pieces on different levels of the 3D pyramid.
- **Connection:** Games focused on connecting pieces across the board or levels, often involving blocking opponents' connections.
- **Patterns:** Games based on forming specific geometric patterns, such as pyramids or other shapes.
- **Completion:** Games in which players strategically place pieces with the objective of filling the board, often involving the tactical placement of balls to block the opponent.
- **Elimination:** Games where players remove pieces or aim to clear the board, often involving strategic removal of balls.
- **Capture:** Games involving the capture of opponents' pieces, often with specific constraints on movement and removal.
- **Counting:** Games that involve some kind of scoring with the winner being the player to satisfy some specified condition the most often.

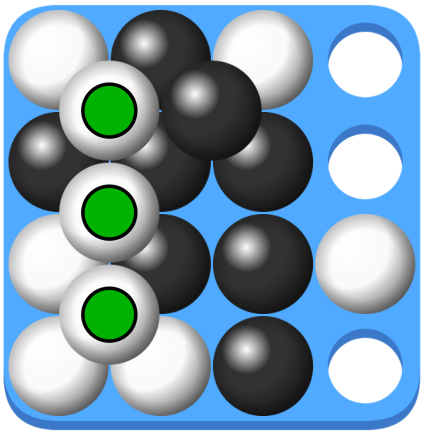

Figure 2.6: Spline game in Ludii won by White

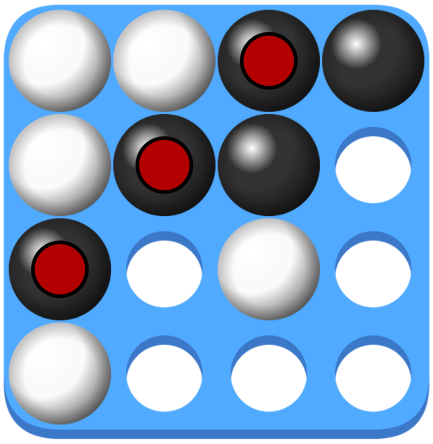

Figure 2.7: Spava game in Ludii lost by Black

- **Puzzles:** Solitaire-style games in which the player must solve specific challenges, often involving optimal placement or movement of pieces.

2.4 Problem Statement

As stated before, integrating 3D games into the Ludii system presents several challenges due to its original design focus on traditional 2D board games. While the Ludii system is highly effective for a wide range of games, it lacks optimised support for modern 3D game mechanics, including those found in the Shibumi set. In particular, Ludii's current language, interface and AI capabilities are not fully equipped to handle the complexities of 3D gameplay, particularly in terms of interaction within a three-dimensional space.

This limitation creates a significant gap in the Ludii platform's ability to accommodate contemporary 3D games, which often require different rule structures and AI strategies. The Shibumi games, as a representation of minimalist yet strategically complex 3D games, provide an ideal framework for exploring these challenges. Therefore, this work aims to extend the Ludii system by adapting it to model, visualise and simulate Shibumi and other 3D games, while at the same time improving AI performance in these environments through the development of new features and techniques.

3

3D (Shibumi) Games in Ludii

The objective of this chapter is to provide an initial insight into how Shibumi games function and can be implemented in Ludii. Therefore, in the first section, we will review the two Shibumi games already implemented in Ludii. The second section will outline the rules and ludemes relevant to this work that define both of these games. As noted in the previous chapter, this will allow us to compare the existing rules and ludemes with the new ones developed throughout this work, which will be introduced in the next chapter. This comparison will further emphasize the necessity of these new ludemes for 3D games.

3.1 Previously Implemented Shibumi Games

In this section, we will examine the two Shibumi games already present in Ludii prior to this work, along with their ludemic construction. This will provide a broader understanding of the state of Ludii concerning these games before our work, as well as a more complete view of how these kind of games function.

3.1.1 Spline

Spline is a two-player Shibumi game that belongs to the N-in-a-Row category (see 2.3 for the categorization of Shibumi games). The players alternate placing a piece (ball) of their own colour (each player having an assigned colour) on any playable point (either an empty hole or a platform made up of 4 other balls). A player wins the game by successfully making a flat line spanning from side to side or from corner to corner on any level of the board. An image of a game won by the White player can be seen in Figure 2.6 above. The game description of *Spline*, which we will analyse in the next section, can be seen bellow in Figure 3.1.

3.1.2 Spava

Like *Spline*, *Spava* is a two-player Shibumi game belonging to the N-in-a-Row category (2.3), where each player alternately places a piece of their colour (with each player assigned a distinct colour) on any playable point (either an empty hole or a platform formed by four other balls). Similarly, a player wins by creating a flat line of their colour that spans

from side to side or corner to corner on any level of the board. The key difference is that a player loses by creating a flat line of their colour that is one short of the spanning size, either orthogonally or diagonally, on any level (only lines of at least two balls counting). An example of a game in which the Black player loses due to this added rule can be seen in Figure 2.7. The ludemic implementation of *Spava* is shown in Figure B.3 in the Appendix.

When analysing the difference between the implementation of Spline and Spava, we quickly notice that the only difference lies in the addition of an end rule, which completely follows the rules of the two games. This allows us, again, to highlight the intuitiveness that the games' ludemic implementation brings and that we presented in the previous chapter.


```
(game "Spline"
    (players 2)
    (equipment {
        (board (square 4 pyramidal:True) use:Vertex)
        (piece "Ball" Each)
    })
    (rules
        (play
            (move Add
                (to (sites Empty)
                    if:(is Flat)
                )
            )
        )
        (end
            (if
                (is Line (- (count Rows) (layer of:(last To))) SameLayer)
                (result Mover Win)
            )
        )
    )
)
```

Figure 3.1: Ludemic implementation of Spline in Ludii
Source: https://github.com/Ludeme/Ludii

One should mention that the version of *Spava* already implemented in Ludii (Figure B.3) differs slightly from the one in the *Shibumi Rule Book* [9] cited in the previous chapter. However, we have now also included the book version, which we will discuss later.

3.2 Ludemes used for Spline and Spava

In this chapter, we will analyze two ludemes used in *Spline* and *Spava*, which are of particular interest due to their adaptations and frequent use throughout the rest of this work. Here, we will describe them in their initial state, while the modifications we have made will be discussed in the next chapter. It is worth noting that, since two Shibumi games are already present, a 2D board representation for Shibumi has also been implemented in Ludii. Consequently, we can utilize this graphical board implementation for the upcoming games without the need to develop a new one. The implementation of a 3D representation could be an interesting endeavour and is discussed in Chapter 6.3.

3.2.1 IsLine

IsLine is a ludeme used to perform tests related to lines, such as determining whether certain pieces form a line of a specified size on a board. It can accept many input parameters (see Figure 3.2 with optional parameters being in brackets with their default value shown at the end of each parameter description), allowing it to be configured for a wide range of line-related tests. By its very nature, *IsLine* is intrinsically linked to the 2D plane. In its initial state, it can perform almost all tests related to lines in two dimensions, which are traversable by a single vector.


```
 * [<siteType>]: The graph element type [default SiteType of the board].
 * <int>: Minimum length of lines.
 * [<absoluteDirection>]: Direction category to which potential lines must belong
   [Adjacent].
 * [through:<int>]: Location through which the line must pass.
 * [throughAny:<region>]: The line must pass through at least one of these sites.
 * [<roleType>]: The owner of the pieces making a line [Mover].
 * [what:<int>]: The index of the component composing the line [(mover)].
 * [whats:{<int>}]: The indices of the components composing the line.
 * [exact:<boolean>]: If true, then lines cannot exceed minimum length [False].
 * [contiguous:<boolean>]: If true, the line has to be contiguous [True].
 * [if:<boolean>]: The condition on each site on the line [True].
 * [byLevel:<boolean>]: If true, lines are detected using the level in a stack
   [False].
 * [top:<boolean>]: If true, lines are detected using only the top level in a stack
   [False].
```

Figure 3.2: Parameters and their descriptions for the `IsLine` ludeme.
Source: [49]

IsLine is of particular interest to us since, as we shall see, it is used in all Shibumi games that belong to the N-in-a-Row category (2.3). However, despite being already adapted for *Spline* and *Spava*, whose rules aim to form lines on a flat plane (and thus in 2D) of the board's pyramid, we will later observe that *IsLine* in its original state is not suited to the different forms a line can take on a 3D board.

3.2.2 IsFlat

IsFlat is a very simple ludeme that takes a simple integer, corresponding to a site on the board, as a parameter. Its function is to check that, in a 3D board, all the pieces in a lower layer are placed so that the piece in the upper layer does not fall off (by fall we are talking here about gravity, a rule not used in Spline and *Spava* but which we will discuss later).

Unlike *IsLine*, the nature of this ludeme is intrinsically linked to 3D, as it verifies that a piece can be superposed on others. However, we will see later that, in the context of 3D interactions, this condition may sometimes not be sufficient and will need to be supplemented.

4

Contributions to 3D Games in Ludii

This chapter examines the contributions of this thesis regarding games and ludemes added or modified in Ludii. The first section offers a brief overview of the games added, avoiding excessive detail. The second section details the modifications and additions made to the ludemes and the rationale behind them. Lastly, the third section provides a concise summary of the tests conducted to validate the new games, ludemes, and ludemic features.

4.1 New 3D Games

In the first subsection, we will briefly present all the games implemented, without providing too much individual explanations or detailing their ludemic implementation. Some of these games will be partially discussed in the second section of this chapter, which introduces the created ludemes, to offer additional context. Complete rules for the Shibumi games can be found in the Shibumi Rule Book [9], as cited earlier. Their ludemic implementations are available on the dev branch of our Ludii repository (https://github.com/cedantoine/Ludii) and will soon be included in the new version of the official Ludii release. Additionally, images of each implemented game in their end-game states are included in the C: Appendix. The second subsection will then briefly address the Shibumi games not implemented in this work and the reasons for their omission.

4.1.1 New Implemented Shibumi Games

As mentioned in Chapter (2.3), Shibumi games are divided into eight categories. This section presents the games implemented, organized by their respective categories, while briefly restating the objectives of the games within each category.

N-in-a-Row

The N-in-a-Row games are games where the objective is to form lines of pieces on different levels of the 3D pyramid. All the games in this category, presented in the *Shibumi Rule Book* [9], have been implemented. They include:

- Spline+
- Splice
- Spree
- Alternative Spava
- Splade
- Sparro
- Sploof
- Spaniel

It should be noted that the *Spline* game presented in the previous chapter (3.1.1) also belongs to this category, but it is not discussed here as it was already implemented prior to this work. Additionally, as mentioned earlier, an alternative version of *Spava* compared to the one presented in the previous chapter (3.1.2), involving neutral red balls, has been added to Ludii as part of this work, which is why it is included here. Finally, it is worth highlighting the game *Sploof*. In this game, the objective is to create a line of four balls of your own colour, visible from the top of the board. The balls do not necessarily need to be aligned on the same plane when viewed from the side of the board. This game provides a great example of how a line can be perceived differently in 3D space. Below, on the left in Figure 4.1, one can observe a finished game of the alternative version of *Spava*, lost by Black. Similarly, on the right in Figure 4.2, a finished game of *Sploof*, won by White, is displayed.

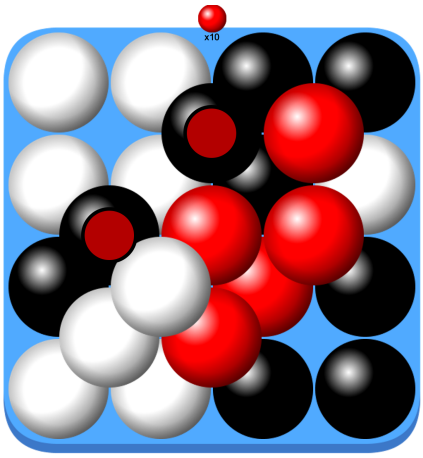

Figure 4.1: Alternative Spava game in Ludii lost by Black

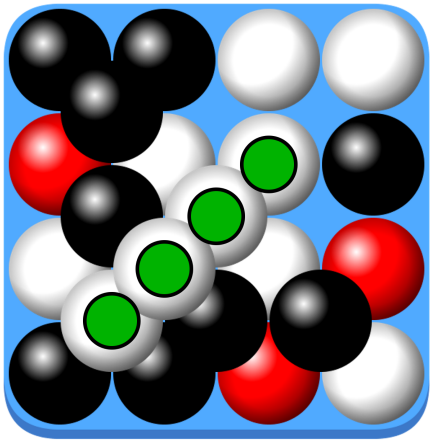

Figure 4.2: Sploof game in Ludii won by White

Connection

Connection games focus on creating connections between pieces across different parts of the board or levels, often while attempting to block the opponent's connections. The *Shibumi Rule Book* [9] presents seven games in this category. Of these, five have been implemented, including:

- Span

- Sponnect
- Spight
- Spice
- Spaji

Two games that exemplify this category are *Span* and *Spice*. In *Span*, the objective is for the White player to establish a group of connected balls spanning from left to right, while the Black player aims to span from bottom to top. An example of a game of *Span* won by the White player is shown in Figure 4.3 below on the left. In *Spice*, each player aims to create the largest group of connected visible pieces. The player with the largest group when the pyramid is completed wins. A game of *Spice* won by the White player is shown in Figure 4.4, below on the right.

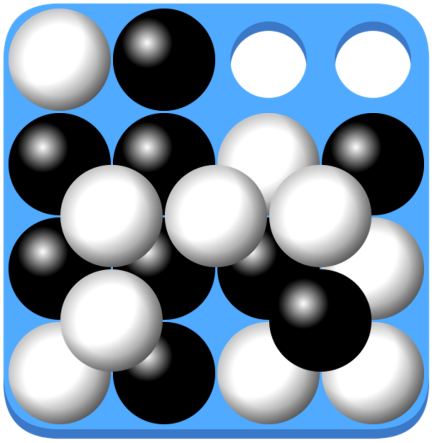

Figure 4.3: Span game in Ludii won by White (connected white pieces spanning from left to right)

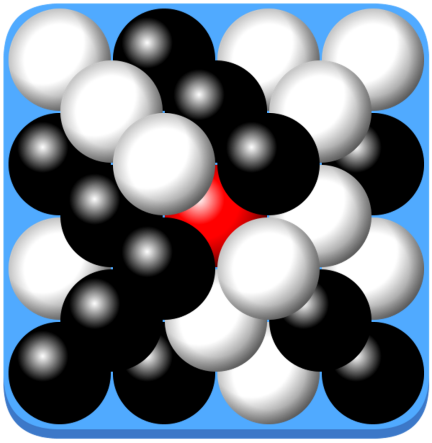

Figure 4.4: Spice game won by White (having a connected group of 8 pieces against 6 for Black)

Pattern

Pattern games involve forming specific geometric patterns, such as pyramids or other 3D shapes. The *Shibumi Rule Book* [9] currently describes only one such game, which we have implemented. It is called:

- Spyramid

Pattern Games, exemplified by *Spyramid*, are intrinsically linked to 3D due to their geometric nature. In *Spyramid*, as the name suggests, the objective is to place five balls of one's own colour at the five vertices of a pyramid (either right-side or upside-down) of any size. Two *Spyramid* game-plays, with victories by the White and Black players respectively, are illustrated in Figure 4.5 and Figure 4.6 below. This type of game can be easily modified by altering the shape of the required pattern. For instance, targeting the four vertices of a cube.

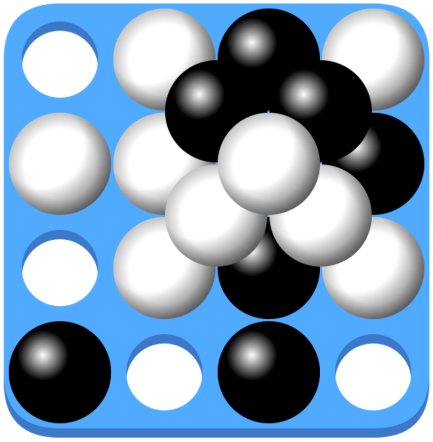

Figure 4.5: Spyramid game won by White (rightside pyramid of size 2)

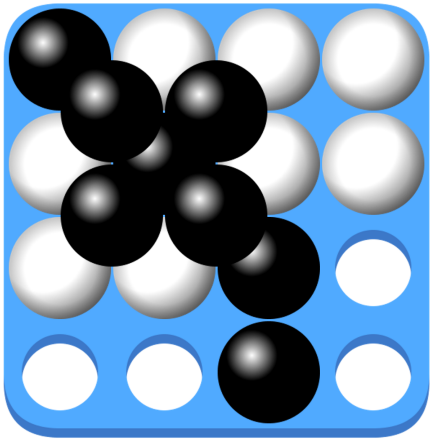

Figure 4.6: Spyramid game won by Black (upside-down pyramid of size 1)

Completion

Completion games are games where players strategically place pieces with the objective of filling the board. These games often involve the tactical placement of balls to block the opponent or score points. The *Shibumi Rule Book* [9] presents three Completion games, all of which were realized during this work. These are:

- Spire
- Spinimax
- Splastwo

Counting

Counting Games are games that involve some kind of scoring with the winner being the player to satisfy some specified condition the most often. The Shibumi Rule Book presents 8 of these games, five of which are implement under this work:

- Sprite
- Spao
- Speedo
- Spirit (of Shibumi)
- Spodd

Two examples of Counting games are therefore *Spirit* and *Sprite*. In the first, players take in turns placing a ball of their colour on any playable point, making sure that no other 'free' ball of the same colour is on the same orthogonal line; or passing if there is no

possible move. The game ends when all players have passed consecutively and the winner is the player with the most balls in play. In the second, Players take turns placing two balls at any playable points, each ball being either the mover's colour or red. As each ball is played, its score is counted: a ball of the mover's colour scores points equal to the number of red balls it touches, while a red ball scores points based on the number of balls of the mover's colour it touches.

Capture

Capture games are games involving the capture of the opponent's pieces, often with specific constraints on movement and elimination. Two capture games are introduced in the *Shibumi Rule Book* [9], both of which have been modelled:

- Spoing
- Spargo

Spargo is a 3D adaptation of the game Go in which pinned pieces remain active even after capture. It is Shibumi's deepest-known game. Players alternate placing a piece of their own colour on the board. After each move, pieces that are no longer free (i.e., no visible connection to a free square on the board) are captured and removed, with the exception of pinned pieces. The game ends when one player has no more legal moves, and the player with the most pieces remaining on the board wins.

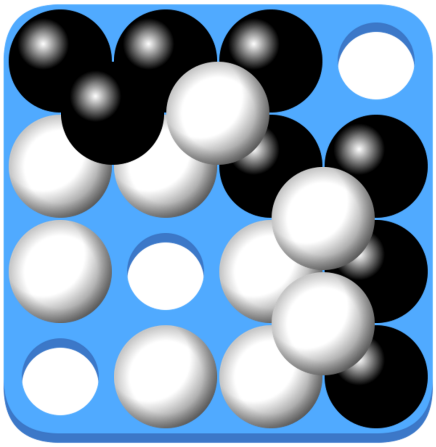

Figure 4.7: Spargo game won by White (9 pieces to 8). Black has no more legal moves where its added piece wouldn't immediately be captured.

Puzzles

Puzzle games are solitaire-type games in which the player has to overcome specific challenges, often by placing or moving pieces in the best possible way. The *Shibumi Rule Book* [9] describes three puzzle games, two of which we have implemented:

- Spuzzle
- Spalone

4.1.2 Not Implemented Shibumi Games

For certain reasons, some Shibumi games have not been made during this work. We will mention them here and explain the reasons why.

Hidden information games

A primary reason that blocked the implementation of three Shibumi games was the presence of hidden information. In Shibumi games, hidden information refers, for example, to a player keeping their stock of remaining balls hidden from the opponent. While such games are, at least, partially feasible in Ludii, a PhD thesis is currently in progress that focuses on a profound modification of Ludii's representation and model for these types of games. This work, part of the collaborative efforts within the GameTable COST Action [50], aims to enable the development of general AI approaches for all imperfect-information games within the platform [51] [52]. It is sensible to wait for its completion before attempting their implementation. The games concerned are:

- Spanic (Elimination category of Shibumi games)
- Spindyzy (Counting category of Shibumi games)
- Spagyric (Counting category of Shibumi games)

Spaghetti

Spaghetti is a Shibumi game in the Connection category. While it does not involve particularly complex mechanics, its victory condition requires evaluating each player's longest strand (i.e., non-branching paths) of balls. An attempt was made to incorporate a strand parameter in the *CountSizeBiggestGroup* ludeme (presented in 4.2.2), which would allow this feature to be added if requested. However, we were unable to implement this without compromising other features of the ludeme itself. This limitation likely necessitates rewriting the entire ludeme to ensure proper adaptation.

Splush

Splush is another Shibumi game in the Connection category. Unlike traditional Shibumi games, the pieces in Splush are not placed directly onto the board. Instead, they are pushed horizontally into any position on the board or onto a flat platform formed by four other balls. When attempting to implement this game, we encountered several challenges with this mechanism. This is another implementation issue that, while likely solvable, would require significant reworking of the *push* ludeme in the Ludii.

Splink

Splink is a Shibumi game of the Counting category. The main challenge we encountered in implementing Splink stems from its gameplay mechanics, which involve assigning specific roles to each player. Due to these roles, the game is played in two phases in its two

existing versions. In one version, players compete in two games, swapping roles between them. In the other, an initial betting phase determines which player takes which role and establishes the victory condition. These multiple variables and stages have made the game's implementation relatively complex and, in our view, not particularly well-suited to Ludii. However, a slight reworking of the game rules could address these issues.

Spin

Spin is a single-player Shibumi game in the Puzzles category. It is a colour-flipping puzzle that does not involve stacking and, therefore, does not utilize Shibumi's 3D property. In addition to being outside the scope of this work due to its lack of 3D mechanics, several features of the game presented challenges during its implementation. The major obstacle was that certain movement effects had to be submitted to decisions. Specifically, in Spin, when placing a piece, depending on the contact it has with pieces of a different colour, it can trigger colour flips. However, in certain very specific configurations, the colour to be flipped is not predictable and is instead subject to the player's choice. Implementing this choice in a so-called Consequence effect, although possible, is a headache in ludemic game implementation.

Other board tilings for Shibumi

Another aspect of Shibumi games that has not been covered in this work is the use of different types of boards shapes for Shibumi-like 3D games. Several of the games discussed above could be adapted to hexagonal or triangular boards, or even multisets of classic Shibumi boards. Nevertheless, a significant reason for not exploring these alternative board types is their limited coverage in the existing literature.

4.1.3 Other Implemented 3D Games

Margo

Margo [53] is the equivalent of Spargo but played on a 6x6 board. Like Spargo, it is a 3D adaptation of the game Go, where pinned pieces remain active even after capture. Players take turns placing their pieces on the board, and after each move, pieces that are no longer free are captured and removed, except for pinned ones. The game ends when a player has no more legal moves, and the winner is the one with the most pieces remaining on the board. The larger board size in Margo introduces additional complexity and strategic depth compared to Spargo.

Furthermore, it is worth noting that a few other 3D games, particularly of the Shibumi style, have already been implemented, with Akron being one such example. Additionally, several unimplemented 3D games are discussed in the 6.1 section.

4.2 Ludeme modifications

For the remainder of this section, it is noteworthy to remember that, in the context of game rules, ludemes can be utilized in three distinct ways:

- Meta: Meta rules are higher-level rules that apply throughout the game and supersede all other rules.

- Start: Starting rules are rules applied at game creation. Most of the time they involve placing pieces in certain regions or sites to give the player a certain starting state.
- Play: Play rules are those that are applied during the game. They indicate what the legal moves are at the current state of the game.
- End: End rules are run after each move to determine whether a game has reached its end state or not. They look for win or lose conditions.

Furthermore, a superludeme is a high-level construct that encapsulates multiple ludemes to define complex game elements or mechanics. It provides a more abstract and generalized way to represent recurring patterns or behaviours, enhancing reusability and simplifying the description of games. This allows for a more efficient and modular approach to game modeling. A part of the initial construction of the superludeme *Is* can be seen in Figure 4.8 bellow.

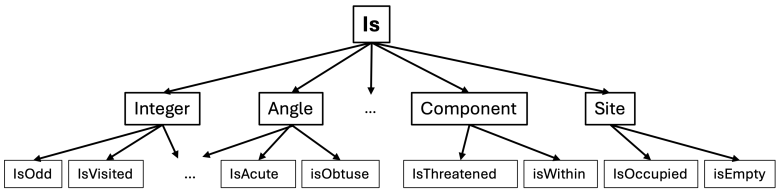

Figure 4.8: Small segment of the Is superludeme structure (missing elements and sub-elements are indicated by three dots)

In this section, we will examine all the modifications made to existing ludemes and the new ludemes added. As mentioned earlier, some games will be referenced to provide additional context. The first sub-section will cover the changes made to existing ludemes in Ludii. The second sub-section will address the added ludemes. Finally, the third and fourth sub-section will discuss the metas and superludemes that have been added and modified.

4.2.1 Updated Ludemes

As mentioned above, in this subsection we will look at the modifications made to the *'classic'* ludemes that existed prior to this work in Ludii.

IsLine

Given its strong connection to the Shibumi N-in-a-Row games category, the *IsLine* ludeme is one of the most frequently reused pre-existing ludemes in this work. While it has already been presented more or less in detail in section 3.2.1, let us briefly recall its usefulness. *IsLine* is a ludeme used to perform tests related to lines, such as determining whether certain pieces form a line of a specified size on a board. This ludeme has a large number of parameters, presented in Figure 3.2, and due to its nature is strongly linked to 2D space.

This ludeme has been modified to adapt it to 3D space by adding three new parameters which are shown in Figure 4.9 below.

```
* [throughHowMuch: <int>]: Minimum number of different piece types the line has to
  be composed of [1].
* [isVisible: <boolean>]: If true, all pieces of the line have to be visible when
  looking from above the board [False].
* [useOpposites: <boolean>]: Wether to use opposites rdials or not when looking for
  lines [True].
```

Figure 4.9: New Parameters and their descriptions for the `IsLine` ludeme

The **first new parameter** to be introduced, *throughHowMuch*, is not directly related to the 3D nature of Shibumi games but rather to an N-in-a-Row game victory condition rule. It indicates that to win, a line must consist of at least two or more different types (in the Shibumi case we speak of colours) of pieces. *IsLine* already included a parameter for defining which types of pieces a line should consist of, but it did not impose a minimum number of each of these piece types. This new parameter is optional in the ludeme and defaults to 1. When specified, it must be referenced by the parameter name (@Name).

The **second parameter** to be introduced, on the other hand, is directly linked to the nature of 3D games. On a three-dimensional board, unlike a 2D board, certain pieces or connections between pieces may be covered by others. The *isVisible* parameter is therefore used to ensure that all pieces of a line are first of all visible, but that all the connections between these pieces are also visible. In Figure 4.10, a line of three black balls is visible, but the connection between two of them (located on sites 18 and 20) is obscured by two white balls (on sites 14 and 24). As a result, when using the *isVisible* parameter, this line would not be counted as having a length of three. As before, this parameter is optional in the ludeme call and defaults to False. Furthermore, when used, it must also be explicitly specified by its parameter name.

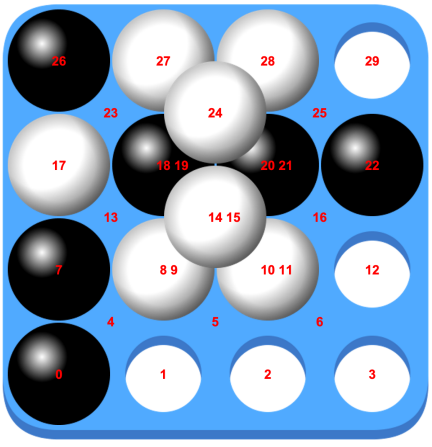

Figure 4.10: Example of a Line-connection hidden by two White pieces

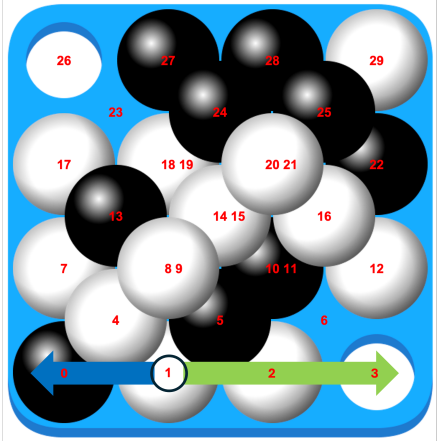

Figure 4.11: Example of Sploof game used for isLine's radial usage explanation

To explain the **third parameter**, it is necessary to provide some background, particularly on radials and their use in *IsLine*.

Radials are imaginary lines extending from a central point in various directions, commonly used to organize or structure elements around that point. In the context of board games, they define the positions or connections of pieces based on their orientation relative to a reference center or piece. As such, radial computation is inherently dependent on the specific board geometry [54].

When *IsLine* searches for a line starting from a specific site, the ludeme considers not only the radials in the direction of the search but also the opposite radials, i.e., those in the direction opposite to the search. For example, consider Figure 4.11. If one searches for a horizontal line passing through the ball at site 1, Ludii will first look for a line in the eastern direction (green arrow on the Figure 4.11), evaluating the radial consisting of sites 2 and 3. Simultaneously, it will evaluate the opposite radial (blue arrow on the Figure 4.11), composed of site 0, as it lies in the opposite direction (west). If the piece at site 0 were white, a line of length 3 could have been detected because of this. As we can see, this feature is very practical for line detection when starting from a given site.

In certain games (e.g., Sparro), it is nevertheless sometimes necessary to determine whether a piece (ball) forms a line with other pieces while being at the extremity of the line. To provide control over the use of opposite radials and enable checking of this specific condition (i.e., considering only one direction and not its opposite), we have introduced the *useOpposite* parameter. This parameter allows *IsLine* to account for opposite radials selectively. It is optional, defaults to True, and must be specified using its parameter name when called.

Computation of Radials

The modification we are going to describe here is not strictly a modification of a ludeme, but rather of a background process; i.e. the way in which the radials, as well as the sites and pieces that compose them, are evaluated. To explain the changes made and the reasons behind them, we will first outline how the radial components are initially calculated.

To evaluate sites (and their associated pieces) along a radial, Ludii begins with a starting site and an initial direction where are looking the radial for. From the starting site, Ludii follows this direction to identify the second site on the radial. To locate subsequent sites, Ludii examines all neighbors of the second site, calculating the angle (in 2D) between the initial direction and the line connecting the second site to each neighbor. If Ludii detects an angle of 0 degrees, it determines that the neighbor lies perfectly on the radial and ceases further examination of that site's neighbors. However, if no 0-degree angle is found, Ludii selects the neighbor with the smallest angle (within a defined maximum threshold) as the next site on the radial. If no angle falls below the threshold, Ludii concludes that the end of the radial has been reached. When a new site is identified, the process is repeated: the new site becomes the starting site, and the line connecting it to the previous site becomes the new direction for further exploration along the radial.

This approach for evaluating sites on radials is highly effective for detecting lines on 2D planes, but it presents a significant limitation when identifying lines in 3D, especially

for lines spanning across multiple layers of the board (keeping in mind that the angle for finding sites on radials is computed in 2D). Referring to Figure 4.12 and 4.13 bellow, suppose we are searching for a line spanning multiple levels and passing through the ball at site 9 (located above site 8). When *IsLine* evaluates the radial formed by sites 4 and 0 (light green arrow), it may erroneously consider the radial formed by sites 14 and 20 (blue arrow) as the opposite radial, since the 2D angle between the two directions is 0 degrees (see Figure 4.12). (Note that it could also have identified the radial formed by site 15 (dark green arrow), at the apex of the pyramid, as the opposite radial, since it also forms a 0-degree 2D angle with the initial direction and is indeed the correct radial direction; however, since *IsLine* stops searching for radial objects as soon as a null angle is detected, it will not consider this second path.) As a result, *IsLine* might incorrectly identify the white balls at sites 4, 9, and 14 as forming a line, even though they are not situated on the same plane.

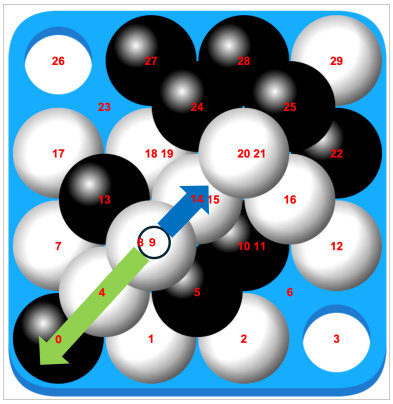

Figure 4.12: 2D view from above of initial radial computation

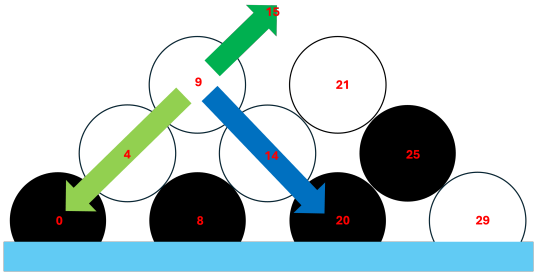

Figure 4.13: Slice view of initial radial computation (initially computes blue arrow but correct computation should be dark green arrow)

A fairly obvious solution to this problem, which we have implemented, is to replace the 2D angle measurement with a 3D angle measurement. This effectively solved the issue of conflicting and erroneous radials, like the one showed in the previous example, where now the radial, composed of site 15 (dark green arrow), is now correctly identified as the opposite radial. However, for one particular game, this solution was not suitable.

Let's recall the Sploof game presented in the 4.1 section. In this game, the objective is to create a line, of length 4, visible from the top of the board, meaning a line seen from a 2D perspective from above of the 3D board, potentially composed of elements at different levels that cannot be traversed by a single vector. One can see such a lines on Figure 4.12 above, composed of balls at 4, 9, 14 and 21, or on Figure 4.14 bellow, composed of balls at 4, 8, 14 and 21. For this game, the new method, which calculates radials as vectors in 3D space, was naturally not suitable, and a custom method for calculating radials had to be implemented. Starting from the initial computation method, the 2D angle measurement for site search was retained, but the condition that the search for other sites should stop

when a radial with a 2D angle of 0 degrees is found was removed. In fact, when searching for radials of this type, a radial site may have several valid neighbouring sites in the same direction, as the z-axis position no longer plays a role. In Figure 4.15 below, two radial computations for this game are illustrated. At site 14, two neighbors are valid as the next radial site. The evaluation of radials for this game thus resembles the construction of a tree, where multiple radials can share the same subset of parts. In the example shown in Figure 4.15, both radials share the subset {0,4,8,14}. However, at site 14, they diverge, proceeding respectively to 20 and 21, before rejoining at {25,29}.

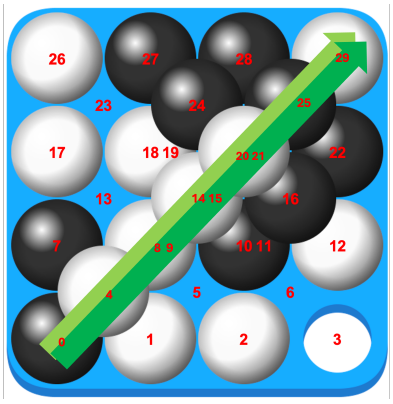

Figure 4.14: 2D view from above for Sploof radial computation

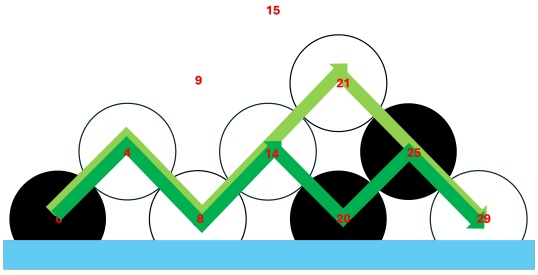

Figure 4.15: Slice view of Sploof radial computation

SitesGroup

The *SitesGroup* ludeme is a ludeme used to list all the sites belonging to a specific group. To define the group membership sites, it takes as parameters the starting position(s) of the group, the conditions for potential pieces to be part of the group, and the direction of connection between the sites in the group. Similar to the *IsLine* ludeme, we have introduced an *IsVisible* parameter to the *SitesGroup* ludeme. As before this new parameter ensures that a piece is both visible and visibly connected to the group on the 3D board. As for *isLine*, this new parameter is optional in the ludeme call and defaults to False. When used, it must be explicitly specified using its parameter name.

SizesGroup

The *SizesGroup* ludeme is used to return the sizes of different groups. To define group memberships and the groups to consider, it takes parameters such as the player's ID or role to include pieces in the group(s), conditions for potential pieces to be part of the groups, the direction of connection between sites in the groups, and the minimum size of each group. Similar to the *IsLine* and *SitesGroup* ludemes, we have introduced an *IsVisible* parameter in the *SizesGroup* ludeme. This new parameter ensures that a piece is both visible and visibly connected to the group(s) on the 3D board. As before, this parameter is optional, defaults to False unless explicitly specified and must be explicitly specified using its parameter name when used.

PlaceRandom

The *PlaceRandom* ludeme is a start ludeme designed to place a defined number (1 by default) of one or more specified pieces types on a region of sites at random. In its basic form, this ludeme iterates through each piece type, in the given order, and randomly places the desired number of that piece on the region. If no more free sites are available in the region, it stops without placing the remaining indicated pieces.

An optional parameter, *randPiecOrder*, has been added to this ludeme, which is set to False by default. This parameter allows you to specify whether the types of pieces to be placed should be selected at random, rather than following the indicated order. It is particularly useful when the number of pieces to be placed exceeds the number of available sites, ensuring that the same pieces are not always selected, and preventing others from always being left unplaced.

4.2.2 Added Ludemes

In this section we will present all the new ludemes we have created as part of this work.

IsFreedom

The *IsFreedom* ludeme takes a region of sites as a required parameter and another individual site as an optional parameter. Neither parameter needs to be called by its parameter name when used.

This ludeme evaluates whether the supplied region has freedom, meaning it checks if the region has a visible connection to an empty square on the board. It is important to note that by "empty square" on the board, within the context of Shibumi, we mean a site on the base layer of the board, not a site placed on other balls. If you want to know if a region will maintain its freedom after placing a piece on a specific site, you supply that site as the optional parameter. In this case, the ludeme will consider the site to be occupied when searching for freedom.

IsPyramidCorners

The *IsPyramid* ludeme is designed to take one or more sites as input parameter and evaluate whether any of the sites form part of a pattern that constitutes the corners of a pyramid structure made of pieces of the same colour. The input sites are optional and default to the last site where a piece was placed if not specified. Furthermore, the parameter does not need to be referenced by its name when called.

This ludeme is particularly relevant in Shibumi games within the Patterns category, where recognizing and interacting with geometric configurations like pyramids enhances the depth of the gameplay. Examples of such detected pyramid patterns can be seen in Figure 4.5 and Figure 4.6. This ludeme, by its nature, is reserved exclusively for 3D boards. However, it could be adapted for use on 2D boards by modifying the pattern to correspond to a 2D shape.

SitesSupport

The *SitesSupport* ludeme is designed to identify and return all sites on the board that have one or more pieces stacked on top of them. It does not take any input parameters, as it always considers all pieces on the board. When applied to a fully filled Shibumi board, it

would return an array of all sites on the board except for the one at the top of the pyramid. Similarly, if applied to a board where pieces are placed only at the base layer, it would return an empty array. This ludeme, like the previous one, is specifically reserved for 3D board games. If applied to a 2D board, it would always return an empty array of sites.

CountSitesPlatformBelow

The *CountSitesPlatformBelow* ludeme takes a site and either a player ID or one or more piece IDs as input parameters. The site parameter is optional, defaulting to the [(to)] site if unspecified, and does not need to be referenced by name. One of the two other parameters is mandatory (only one can be used at a time) and must be explicitly named.

This ludeme returns the number of sites beneath the specified site occupied by the given piece IDs or pieces belonging to the specified player. It is reserved for 3D platforms and returns 0 on 2D boards. For example, applying it to site 24 in Figure 4.10 with white pieces would return 2.

CountSizeBiggestLine

The *CountSizeBiggestLine* ludeme, as its name suggests, returns the size of the largest line on the game board whose components satisfy specific conditions. The ludeme takes two input parameters: the direction in which the parts of the line must be connected and the condition(s) that each part must meet to be included in the line. Both parameters are optional. The direction defaults to Adjacent, and the default condition only checks if the part is occupied by a piece. The first parameter, when used, does not need to be referenced by name, while the second must be explicitly specified by its name.

While this ludeme is not exclusively tied to 3D board games, its necessity arose in certain 3D games where pieces can be removed or moved, potentially causing other pieces to shift (e.g., falling due to gravity). In games focused on forming lines, such moves may result in the creation of one or more allied or enemy lines that are not necessarily directly connected to the site of the initial move. This creates the need to identify the longest line formed by a move across the entire board.

CountSizeBiggestGroup

Similar to the *CountSizeBiggestLine* ludeme, the *CountSizeBiggestGroup* ludeme returns the size of the largest group on the board whose pieces satisfy specific conditions. It takes the same arguments as *CountSizeBiggestLine*, with the addition of an *isVisible* parameter. When used, this parameter ensures, as in *isLine* and *SitesGroup*, that each piece in the group is visible and visibly connected to the group. The default values and call conditions for the first two parameters are the same as those in *CountSizeBiggestLine*. The isVisible parameter, on the other hand, must be explicitly referenced by its parameter name when called. It is optional and defaults to False.

While this ludeme is also not exclusively linked to 3D games, its necessity arises from the same context as *CountSizeBiggestLine*: the potential creation of one or more new groups when pieces cascade due to the movement of a single piece on a 3D board. This creates the need to identify the largest group across the entire board after a single move.

4.2.3 Modified Superludemes

In the creation of the ludemes discussed earlier, we utilized three superludemes: The *Is* superludeme for *IsSitesPyramid* and *IsFreedom*; the *Count* superludeme for *CountSitesPlatformBelow*, *CountSizeBiggestLine*, and *CountSizeBiggestPlatform*, and finally the *Sites* superludeme for *SitesSupport*.

Each ludeme we created is invoked through its associated superludeme, which subsequently redirects the call to the corresponding ludeme. In Figure 4.16, we present the simplified new graph associated with the *Count* superludeme. The graphs for the *Is* and *Sites* superludemes are provided in Figures D.29 and D.30 in the Appendix.

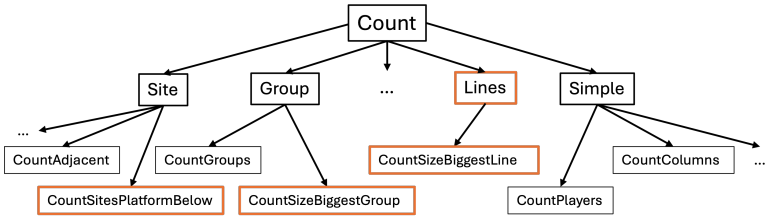

Figure 4.16: Simplified graph associated with superludeme Count (missing sub-elements are indicated by three dots and red counters highlight the elements created during this work)

4.2.4 Added/Modified Meta Ludemes

As previously discussed, meta rules are overarching principles that govern the entire game, taking precedence over all other rules. Within the scope of this work, we have developed two meta ludemes, both of which are centred around the concept of gravity. These meta ludemes will be introduced and explained in detail below.

NoStackOnFallen

The first meta to be created is *NoStackOnFallen*. As explained in Section 3, the highly useful *IsFlat* ludeme needed to be extended in certain scenarios, and this meta provided the solution. As previously mentioned, in some 3D games (notably Shibumi games), certain pieces on which others rest can be moved, causing the supported pieces to shift or fall on the board. In most games that allow this, a rule prohibits placing the piece that caused the fall onto the pieces that fell, as this could lead to endless cyclical turns.

Therefore, to complement the *IsFlat* rule—where the requirement for a flat surface on which to place a piece is no longer sufficient—we introduced *NoStackOnFallen*. This ensures, as its name suggests, that when placing a piece, it does not rest on any piece whose fall it caused.

It is also worth noting that, to create this meta, we first developed the *NoStackOn* superludeme, which encompasses the *NoStackOnFallen* ludeme.

PinSupportMultiple

The *PinSupportMultiple* meta already existed in Ludii prior to this work. This meta ensures that a piece directly supporting multiple other pieces cannot be removed. However, a piece

supporting only one other piece (in direct connection) can be removed. For example, in Figure 4.10, the piece at site 28 cannot be moved because it supports the pieces at sites 24 and 25. In contrast, the piece at site 29 can be moved, as it only supports the piece at site 25 in direct connection.

As part of this work, we made a slight modification to this meta so that it also applies to the displacement of pieces directly supporting more than one piece, not just their removal. In all Shibumi games, a piece supporting multiple others cannot be moved or removed. This restriction exists because, when multiple pieces rest on the same piece, removing it introduces randomness: the piece replacing it is subject to chance, as each of the directly supported pieces has an equal probability of falling.

4.3 Validation Tests

To implement the various ludemes and games, extensive testing was essential to ensure their proper functionality. This was achieved through a combination of integrity tests for the games and JUnit tests for the ludemes. These tests are crucial to the project's success because ludemes are dynamic objects that can be modified or enhanced over time. Since changes to a ludeme can potentially introduce errors, these tests act as a safeguard, providing immediate feedback on any unintended consequences of updates. The testing framework we employed ensures the overall integrity of the system, allowing us to maintain its robustness throughout the development process.

4.3.1 Integrity tests

For each game we have implemented, one or more trials have been added. These trials are sequences of moves made in accordance with the game's rules, which may or may not result in a final game state. Executing these trials ensures the integrity of a game when its ludeme implementation is modified or updated, confirming that all moves can still be performed correctly and that the changes do not negatively impact the game.

These trials also serve an additional purpose. When a specific ludeme is modified, the trials of games that utilize this ludeme can verify that the changes do not introduce unintended consequences, such as making certain moves impossible. In this way, the trials help ensure that ludemes interact correctly across the diverse scenarios encountered during gameplay, even if this is not their primary purpose.

Moreover, it is worth mentioning that, since the Ludii repository is hosted on a public GitHub, these trials are executed automatically after each push to validate the integrity of the modified code.

4.3.2 JUnit Tests

In developing the ludemes, we adopted the Test-Driven Development (TDD) methodology [55]. This approach entails designing tests prior to the implementation of the ludemes. These tests rely on predefined game contexts on which the expected outputs of the ludeme evaluations are known in advance. Once the tests have been defined, the ludeme is implemented and validated by confirming that its evaluations on the predefined contexts yield the expected results.

To construct these predefined contexts, we employed a Shibumi game board with ball colours specifically tailored to the ludeme under evaluation. Using a simplified Shibumi game description, the balls were arranged on the board in configurations relevant to the ludeme being tested. Once the desired context (i.e., the ball layout on the board) established, the ludeme can be applied, and its output can be compared against the expected value to ensure accuracy and correctness.

4.4 Observations during Ludeme and Game Creation

During this thesis, we implemented 26 new Shibumi games as-well as one other 3D game, modifying 5 existing ludemes, creating 7 new ones and modifying radial computation to integrate 3D features. Interestingly, the creation or modification of just 1–3 ludemes often enabled the development of up to 5 new games, clearly highlighting the necessity and utility of these additions. Furthermore, we observed that many of the changes were driven by common interactions specific to the 3D nature of the board, such as shifts in piece visibility due to vertical positioning or chain reactions caused by the movement of lower pieces. These findings emphasize the significance of 3D-specific dynamics in ludeme and game design. In Table 4.1 below, all games are listed along with the ludemes that were either added (highlighted in red) or modified (highlighted in blue) to enable their implementation.

Table 4.1: Table linking the implemented games and modified/added ludemes used to implement them

Game	Modified or Added Ludemes used
Spline+	PinSupportMultiple + NoStackOnFallen + CountSizeBiggestLine
Splice	IsLine
Spree	IsLine
Alternative Spava	/
Splade	IsLine
Sparro	IsLine
Sploof	PinSupportMultiple + Taylor-made Radial Computation
Spaniel	IsLine
Span	SitesGroup
Sponnect	/
Spight	PinSupportMultiple + NoStackOnFallen
Spice	PinSupportMultiple + CountSizeBiggestGroup
Spaji	CountSizeBiggestGroup
Spyramid	PinSupportMultiple + NoStackOnFallen + IsPyramidCorners
Spire	CountSitesPlatformBelow
Spinimax	CountSitesPlatformBelow

Game	Modified or Added Ludemes used
Splastwo	/
Sprite	/
Spao	/
Speedo	PinSupportMultiple + SizesGroup
Spirit (of Shibumi)	SitesSupport
Spodd	/
Spargo	SitesSupport + IsFreedom
Spoing	/
Spuzzle	CountSizeBiggestGroup
Spalone	PlaceRandom + CountSitesPlatformBelow

5

AI Agents for Shibumi Games

Having implemented several 3D games, primarily Shibumi, it was time to investigate and attempt to improve the performance of AI agents in these games. This chapter examines the performance and development efforts undertaken for certain agents as part of this work.

As explained in the previous chapter, given Ludii's extensive library of games, the AI agents implemented on the platform are primarily designed to operate across a wide range of games and their variants [46].

This chapter is divided into three sections. The first section briefly presents the main agents implemented in Ludii that we have worked with. The second section outlines the methodology used to compare and improve the performance of the studied agents. Finally, the last section discusses the various results obtained and the improvements made.

5.1 Overview of Ludii's Major AI Agent/s

As previously mentioned, most of the agents implemented in Ludii are designed to be applicable to a wide variety of games. Therefore, a majority of these agents are search-based, meaning they rely on algorithms to explore possible future game states and determine optimal actions. A classic example of a search algorithms, widely regarded as one of the most popular approaches in GGP and extensively used by agents in Ludii, is the Monte Carlo Tree Search (MCTS) family and its numerous variants [47]. One can also think of more classical search algorithms, such as Alpha-Beta pruning [56].

In this subsection, we will briefly explain how MCTS works, as it is not only the most predominant algorithm among Ludii's agents but also the one we have relied on the most in our experiments. Furthermore, we will also briefly discuss Alpha-Beta pruning, as it was also used in our experiments. This will allow us, first, to compare both search methods and later maybe to understand why one or the other performs better in specific contexts.

5.1.1 Monte Carlo Tree Search & Variants

The family of algorithms known as MCTS [47] relies on two fundamental principles. First, the true value of an action can be approximated through random simulations. Second, these values can be leveraged to guide the policy towards a best-first strategy. Based on these principles, the algorithm iteratively constructs a partial game tree, each node representing a state in the domain, informed by the results of previous explorations of this tree. The basic MCTS algorithm iteratively creates this tree until a predefined computational budget, such as time, memory, or iteration constraints, is reached. It does so by iteratively applying four steps, depicted in Figure 5.1, at each iteration [57].

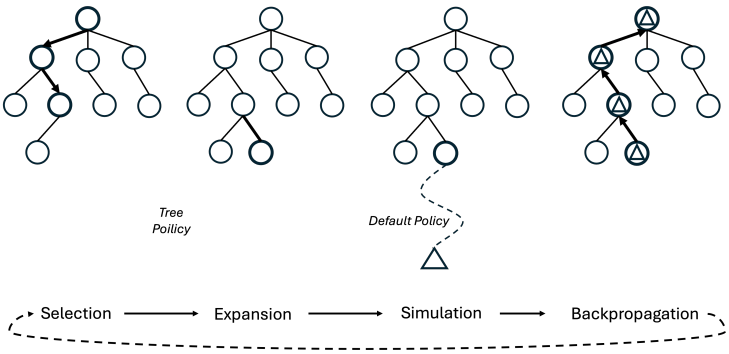

Figure 5.1: MCTS Iteration process [47]

- *Selection*: Starting from the root node, a child selection policy is applied recursively down the tree until the most urgent child for expansion is identified (a node is considered expandable if it represents a non-terminal state and has unvisited children).
- *Expansion*: One or more nodes are added from the selected child, according to the possible actions.
- *Simulation*: A simulation, based on the default policy, is carried out using the new nodes to produce a reward value Δ.
- *Backpropagation*: The outcome (Δ) of the simulation is backpropagated through the previously selected nodes in order to update their statistics.

The four steps of MCTS, presented above, can be grouped into two distinct policies: the *tree policy* and the *default policy*. The tree policy groups together the selection and expansion steps, by iteratively selecting and/or creating one or more new leaf nodes in the existing search tree. The default policy operates during the simulation stage, playing the domain from a given non-terminal state to calculate a value estimate. The backpropagation stage does not use a policy itself, but updates the statistics of the parent nodes to guide future decisions about the tree's policy. When the search is interrupted or the computation budget is exhausted, the algorithm selects an action from the root node using a predefined criteria. Some examples include selecting the maximal child node (the child node with the

highest reward), the robust child (the most visited child), the maximal robust child node (a child with both the highest reward and the highest number of visits, or continuing the search if none exists), or the safe child node (the child that maximizes a lower confidence bound) [58] [59].

Upper Confidence Bounds for Trees (UCT)
Upper Confidence Bounds for Trees (UCT) [47] is one of the most widely used and effective variants of MCTS. UCT utilizes the UCB1 (Upper Confidence Bound 1) [60] method as its tree policy, which has been shown to be both simple and effective in balancing exploration and exploitation during the search process. A child node is thus selected to maximize:

$$\mathrm{UCT}(v) = \bar{X}_j + 2C_p \sqrt{\frac{2\ln n}{n_j}}, \tag{5.1}$$

where $\bar{X}_j$ is the average reward of node j, n is the total number of visits to the parent node of j, n_j is the number of visits to node j and C_p s a constant that controls the exploration-exploitation trade-off.

When multiple child nodes share the maximum value, ties are typically resolved by random selection [61]. The remainder of the algorithm proceeds as outlined previously in classical Monte Carlo Tree Search (MCTS).

MCTS with Move Average Sampling Technique (MAST)
MCTS with Moving Average Sampling Technique (MAST) [47] is a variation of the MCTS algorithm designed to improve the accuracy and efficiency of action value estimation. It is particularly effective in scenarios with large action spaces or noisy rewards. MAST enhances the basic MCTS framework by incorporating a moving average to estimate action values, which smooths fluctuations in reward estimates and facilitates faster, more reliable convergence.

In MAST, a table is maintained to record the average reward for each action, regardless of the state in which it is taken. This table is updated during the backpropagation step based on the rewards obtained from simulations. During subsequent simulations, these averages are used to guide action selection, with a Gibbs distribution favoring more promising moves. This technique helps to mitigate the influence of extreme values in the reward distribution, resulting in more stable and consistent decision-making.

MAST therefore primarily distinguishes itself from other MCTS variants through its unique backpropagation method, which utilizes simulation results in a very specific manner.

MCTS with Generalized Rapid Action Value Estimation Estimation (MC-GRAVE)
Monte Carlo Tree Search with Generalized Rapid Action Value Estimation (MC-GRAVE) [62] is a variation of the classic MCTS algorithm designed to prioritize exploiting existing data over exploring new actions.

In standard MCTS, the selection step strikes a balance between exploring unvisited moves and exploiting knowledge from previous simulations. MC-GRAVE modifies this selection approach to focus predominantly on exploitation. Rather than actively experimenting with unexplored possibilities, it selects moves based on data from well-explored

ancestor states. If the current state has insufficient exploration (i.e., fewer simulations than a predefined threshold, referred to as *ref*), MC-GRAVE references the nearest ancestor state with adequate data. This ancestor state then provides the statistics used to estimate the best move.

MC-GRAVE builds upon concepts from RAVE (Rapid Action Value Estimation) [47] and AMAF (All Moves As First). Both RAVE and AMAF accelerate learning in MCTS by using statistics from moves encountered in simulations beyond the exact path taken. While RAVE blends move value estimates from both the current state and ancestor states, MC-GRAVE further refines this by exclusively using data from ancestor states that exceed the ref threshold, ensuring more reliable exploitation of prior knowledge.

5.1.2 Alpha-Beta pruning

The Alpha-Beta pruning algorithm [56] is an optimization of the Minimax algorithm, aimed at making it more efficient by reducing the number of nodes that need to be evaluated. This is achieved by skipping over branches of the game tree that cannot affect the final decision. The technique relies on two key principles: pruning and the minimax decision rule. During the process, two values, alpha and beta, are maintained. Alpha represents the best score achievable for the maximizing player so far, while beta represents the best score for the minimizing player. If a node's value lies outside the current alpha-beta range, further exploration of that branch becomes unnecessary. This pruning significantly reduces computational overhead. The algorithm works recursively, exploring the game tree from root to leaves. At each step, it applies the minimax decision rule while systematically pruning branches that are guaranteed to be irrelevant to the final outcome. A simplified example of this process is shown in the Figure 5.2 below. In this scenario, the white agent aims to maximize the score, while the black agent tries to minimize it. Here, the exploration begins at the leftmost leaf node. By updating and maintaining the alpha and beta values throughout the process, the algorithm can identify branches that will not affect the final decision. As a result, in our example, it avoids evaluating four nodes.

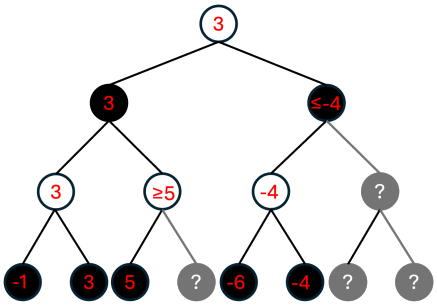

Figure 5.2: Alpha-Beta pruning example

Nevertheless, Alpha-Beta pruning strongly relies on heuristics to evaluate non-terminal game states, especially in large-state games where exploring the entire tree to terminal nodes is computationally infeasible. Heuristics provide approximate evaluations that enable Alpha-Beta to prune unpromising branches and focus on the most relevant parts of the

tree. In contrast, Monte Carlo Tree Search (MCTS) uses random simulations to guide its search, making it less dependent on domain-specific heuristics. Without a good heuristic, Alpha-Beta struggles to perform efficiently in complex games, highlighting its reliance on domain knowledge.

5.2 Methodology for Performance Evaluation and Improvement

The methodology used to experiment with AI agents in this work builds upon existing implementations of Ludii agents. The objective was to identify the most effective agents and potentially enhance them by incorporating tailor-made heuristics, rather than developing agents entirely from scratch. Consequently, the methodology we followed is divided into two parts. In addition, it is important to note that the experiments were conducted on a subset of games[1] of the Shibumi games implemented in this work (N_in_a_Row and Connection category), as not all of the games had been implemented when we began working on this AI aspect.

In the **first phase** of experimentation, due to the lack of existing heuristics adapted to 3D, we initially decided to compare different agents without providing them with heuristics. Based on the depth and average branching factor of the Shibumi games, we selected 5 agents, including one random agent. Each of these agents was pitted against the others in the 3D game set implemented at the time, with 4 distinct reflection times (0.25s, 0.5s, 1s, 2s) between moves and no reflection time before the games. Each configuration "game, pair of agents, reflection time between moves" was simulated 100 times.

In the **second phase**, based on the results of the first phase and considering that some agents are potentially more disadvantaged by the lack of heuristics than others, we decided to select one agent that stood out from the rest and was suitable for the use of heuristics. The goal was to improve this agent by applying existing heuristics and creating one or more relevant new ones. We then pitted the agent in its initial state against the ones equipped with a heuristic. These pairs were tested with the added focus of observing the differences in results according to the type of 3D game. This aimed to establish a link between the type of game and the heuristics used.

All the results presented in this work, unless otherwise specified, were obtained realising simulations on the Lemaitre4 cluster provided by CISM/UCLouvain and CÉCI. This cluster is equipped with 5120 Genoa 2.4 GHz CPUs and 766 GB of RAM [63].

5.3 Results and Achievements in AI Agent Optimization

In this subsection, we will present the results obtained, and the decisions made during the implementation of the methodology described above in the here above subsection 5.2.

[1] Spline+, Splice, Spree, Spava, Splade, Sparro, Spaniel, Span, Sponnect, Spight, Spice

5.3.1 First experimental Phase

The initial step in our process involved analyzing the state space of Shibumi games. We therefore conducted 500 simulations, with random moves, for each of the 11 games studied, using a Mac Pro M3 laptop. During these simulations, we recorded the average branching factor and depth for each game. Using these results we thus estimated that, on average, these games have a branching factor of 12.44 and a depth of 23.12 moves. Thus leading to an estimated number of states of $12.5^{23.12}$.

It is interesting to note that the branching factor across different Shibumi games can vary significantly depending on the game's mechanics. Games that allow the removal or movement of pieces already on the board, in addition to simply adding them, tend to have a much larger state space. For example, in Spline+, we found an average branching factor of 37.76 and an average depth of 45,79 moves. In contrast, a more traditional game like Splice, which only allows the addition of balls, has a smaller average branching factor of 20.5 and a depth of 10.85.

Upon analyzing this state space, we quickly realized that Alpha-Beta pruning, having proven effective on mid-sized state spaces (i.e compared to Go or Chess), could be a promising approach and included it in our first set of agents.

However, recognizing that Alpha-Beta pruning is often limited by the absence or poor adaptation of heuristics due to its highly domain-dependent nature, we decided to also incorporate the classic implementation of the Monte Carlo Tree Search (MCTS) algorithm, UCT [47], along with two of its variants: MAST (Monte Carlo Tree Search with Move Average Sampling Technique)[62] and MC-Grave (Monte Carlo Tree Search with Generalized Rapid Action Value Estimation Estimation) [47], all presented above. Furthermore, all three known for their ability to efficiently explore large search spaces while being less reliant on specific domain knowledge.

All three games were used in the manner previously described in the section 5.1.1. For MAST, the selection policy employed was UCB1, as with UCT. In both cases, we set the constant $C_p = 1/\sqrt{2}$ [61]. Additionally, for MC-GRAVE , we set the reference variable *ref* to 100 loosely based on hyperparameter tuning of [62]. Furthermore, we employed a robust child action selection criterion, as described in the section 5.1.1 above for all three methods.

As last agent, we included a random agent in our experiments to establish a baseline for measuring the efficiency of the other agents. This efficiency benchmark provides a point of comparison, allowing us to highlight the strengths and weaknesses of the more sophisticated algorithms.

The results of the simulations involving these 5 agents, as described in 5.2, are presented in Figure 5.3. These results summarize the performance across all the games tested with the thinking time of 1.0 seconds between moves.

Observing Figure 5.3, it is clear that UCT and MAST outperformed the other agents. Furthermore, the standard deviation of the results between the four non-random agents, being 0.08, further strengthened this intuition.

A possible explanation for the poorer results of MC-Grave and Alpha-Beta lies in their inherent limitations. MC-Grave estimates action values quickly based on simulation

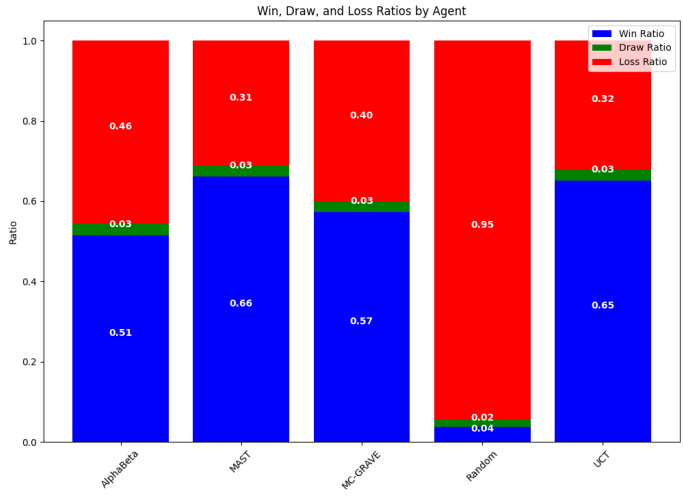

Figure 5.3: Win-Draw-Loss ratio of initial 5 Agents with 1.0 sec thinking time on all Games

outcomes, assuming stable payoff distributions across states. However, in Shibumi, on can intuitively assume that the value of a move is highly dependent on positional context, what is strong early in the game may lose relevance later, or vice versa. Alpha-Beta, on the other hand, relies heavily on domain-specific heuristics, and in their absence (as in our case), it depends on random decisions during exploration and must evaluate moves to the leaf nodes. This can result in computational overload, reducing its ability to explore all actions thoroughly and compromising its strategic effectiveness.

Comparing MAST and UCT, it quickly became clear that UCT should be used to optimize the heuristics for the remaining experiments. MAST employs the most average sampling technique for its playout generation. Replacing this playout generation with a heuristic would closely resemble using UCT with a heuristic, rendering such a substitution pointless.

5.3.2 Second experimental Phase Setup

Once we had decided on the agent to test our heuristics (i.e., UCT), we still needed to determine which heuristics to use and how to integrate them into the agent. Regarding the heuristics, we selected five already implemented in Ludii, chosen for their relevance to the nature of Shibumi games, and we also implemented an additional heuristic, *TopLayerProximity* (presented below), which utilizes the 3D feature of the games. All six heuristics are presented below.

centreProximity

centreProximity is a heuristic that, as its name suggests, favours states with a greater number of pieces in the centre of the board. It is a 2D measure, where the score of a state for a given player is calculated by summing the value of each of the player's pieces

(in Shibumi, all pieces have the same value, i.e., 1), multiplied by the inverse ratio of the distance ($1 - distance/distanceMax$) from the 2D centre of the board (i.e., the piece's distance from the centre when viewed from above). As a result of this calculation, the score increases as more pieces are placed near the centre. This heuristic can be quite interesting in our case, as pieces in the centre of the board potentially interact more with others than those positioned at the edges.

lineCompletionHeuristic

lineCompletionHeuristic is a heuristic that favours game states where lines of connected pieces are closer to completion. It works by evaluating potential lines along radial paths extending from each piece's position. The score for a state is determined by the length of these lines and how many pieces are already in place. The heuristic gives more weight to lines that are nearly complete, rewarding longer lines with more pieces. It also considers whether these lines are blocked by the opponent. This heuristic is particularly relevant for us in the N_in_a_Row category of Shibumi games, where the objective often revolves around creating lines of pieces.

cornerProximity

cornerProximity is a heuristic that favours game states where pieces are positioned closer to the corners of the board. Similar to *centreProximity*, it works by calculating the 2D distance of each piece, but this time from the corners, giving more weight to pieces that are closer to one of the corners. The score for a state is based on the weighted sum of each piece's proximity to the closest corner, calculated as ($1 - distance/distanceMax$). This heuristic can be particularly useful in Shibumi games, where visibility and manoeuvrability of pieces are important, as the corner pieces of each layer are always visible and can be displaced if the game rules allow (since no more than one piece can be stacked on top of them, meaning they are never pinned).

sidesProximity

sidesProximity is a heuristic that favors game states where pieces are positioned closer to the sides of the board. It works similarly to cornerProximity and centreProximity, but instead of considering the distance from the corners or center, it calculates the 2D distance from the nearest side of the board. The score for a state is determined by the weighted sum of each piece's proximity to the closest side, calculated as ($1 - distance/distanceMax$). This heuristic could be useful in Shibumi games where visibility and controlling or positioning pieces near the edges or sides may offer strategic advantages.

nullHeuristic

nullHeuristic is a heuristic that essentially does nothing, its value for any game state is always zero, regardless of the positions of the pieces. It is used as a baseline or placeholder when no specific heuristic is applied. The score does not change based on the game state or the player's position, and it is typically used for testing or situations where no particular strategy is desired. In practice, the *nullHeuristic* serves as a way to eliminate any influence from a heuristic when experimenting with other aspects of the game or the agent's decision-making process. In our case, we used it as a baseline to compare the effectiveness of other heuristics.

topLayerProximity

topLayerProximity is the first heuristic that takes advantage of the 3D nature of Shibumi games and was specifically implemented in this work. It evaluates game states by considering the layer each piece is in, with a particular emphasis on pieces positioned higher in the 3D structure. The heuristic works by assigning a score to each piece based on its layer, with higher layers receiving a greater score.

For each piece, the weight is multiplied by the layer it occupies, and this value is summed to calculate the overall score. This means that pieces on higher layers are favoured, as they are given more weight in the heuristic's evaluation.

In Shibumi games, the 3D nature of the board is crucial for determining the strategy, as higher layers can often provide better visibility and manoeuvrability. This heuristic takes advantage of that aspect by rewarding pieces in elevated positions, making it unique among the other heuristics that focus on the 2D board layout. Its implementation in this work allows the agent to make more informed decisions based on the additional complexity introduced by the 3D nature of the game.

Now that we have presented the heuristics, let's discuss how they have been used in UCT. In order to get a real feel for the capability of each heuristic, we decided to integrate each heuristic into UCT using the AlphaGoZero backpropagation approach. This involves setting the *playoutValueWeight* to zero and estimating the value of each node not by referencing random playouts generated from that node, but by directly using the value attributed to the state of the node by the heuristic. This value is then backpropagated through the tree during the UCT search process.

Concerning the simulations carried out, as mentioned above, we conducted 100 simulations for each UCT agent equipped with a heuristic against the classic UCT agent without heuristics, which was used in the first experimental phase. For reasons of efficiency, and having found no significant correlation between reflection time and win ratio between agents in the first phase, we performed these simulations with a fixed reflection time of 1 second. This approach allowed us to focus on comparing the performance of the heuristics without introducing additional variables related to time complexity, ensuring a more controlled and consistent evaluation.

5.3.3 Second experimental Phase Results

Hereunder, we will now analyze the results obtained by considering both of the Shibumi game categories on which the agents were studied. This will allow us to evaluate how each heuristic performed in the context of these different game types.

Heuristic results on N_in_a_Row games

The results of UCT agents equipped with their respective heuristics against the standard UCT agent, based on play-offs, playing on Shibumi N_in_a_Row games, are shown in Figure 5.4 below.

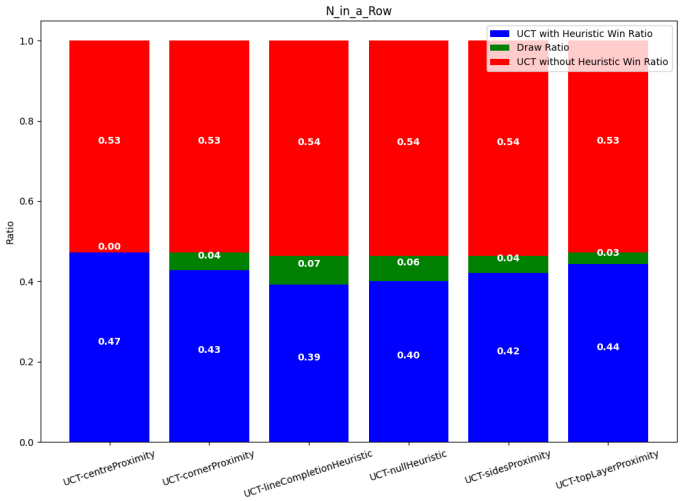

Figure 5.4: Heuristic equipped UCT agents' performance compared too classical UCT on Shibumi N_in_a_Row games

We can clearly observe in Figure 5.4 that no heuristic stands out significantly, and none manages to consistently outperform the classic UCT without heuristics (based on the playouts). Several factors could explain this outcome.

The first one is that no simple heuristic may be capable of adequately modeling the dynamics of this type (3D Shibumi games) or category (N_in_a_Row) of game. Our heuristics would then systematically fail on a majority of the Shibumi games. Another possible explanation is that, while the games belong to the same category, they may not share the same favourable strategies or actions. Consequently, a single heuristic applied uniformly across these games would struggle to deliver consistently superior results.

With this in mind, we investigated the *lineCompletionHeuristic* in greater depth, given its underwhelming performance despite its seemingly natural alignment with the objectives of the N_in_a_Row category. By examining the results of this heuristic on individual games, it quickly became evident that the second explanation appeared to be the root of the problem.

For instance, in a game like *Spline+*, the heuristic significantly outperforms the classic UCT, winning nearly 85% of games. This outcome aligns with the game's goal of creating flat lines. However, in a game like *Spava*, the heuristic almost always loses. In Spava, while the goal also involves creating a flat line, the approach is crucial, making an intermediate line shorter than a layer's length immediately results in a loss. As a result, the heuristic's guidance probably often led to losing positions in such cases.

With this in mind, we revisited the rules of the various Shibumi games in the N_in_a_Row category and realized that, although all these games share the objective of creating lines, the methods for achieving these lines vary significantly. Factors such as combining different types of pieces, forming lines across different levels, and other specific mechanics differ greatly among the games. As a result, a simple heuristic like prioritizing flat lines struggles to accommodate the diverse mechanisms involved in line formation across the category.

We therefore concluded that a more effective approach would have been to design heuristics tailored to each individual game rather than attempting a one-size-fits-all solution for the category. Alternatively, another promising direction would be the creation of a complex heuristic capable of accounting for multiple line-formation possibilities across games. Both of these approaches present interesting avenues for future work.

Heuristic results on Connection games

The results of UCT agents equipped with their respective heuristics against the standard UCT agent, based on play-offs, playing on Shibumi Connection games, are shown in Figure 5.4 below.

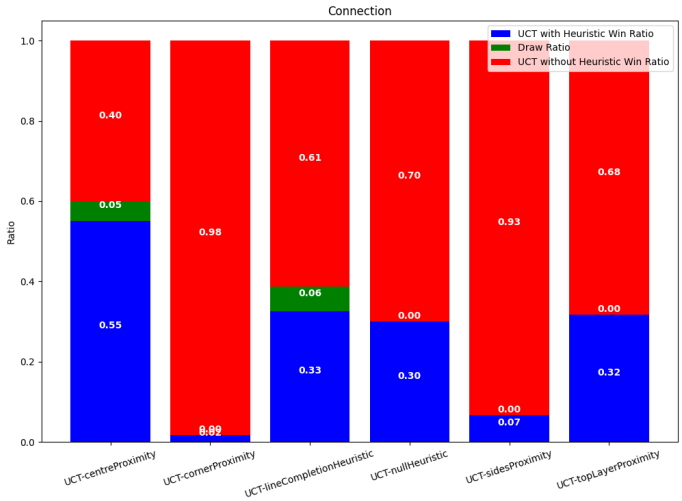

Figure 5.5: Heuristic equipped UCT agents' performance compared too classical UCT on Shibumi Connection games

Unlike the results observed in the N_in_a_Row games, the Figure clearly shows one

heuristic, *centreProximity*, standing out and even outperforming the classic UCT by a significant margin on the Connection games. Upon analyzing these results in conjunction with the rules of these games, this outcome becomes immediately apparent. Indeed, all the connection games studied share the objective of connecting specific areas of the board or forming one or more large groups.

On the Shibumi board, the central squares are those with the highest number of potential connections to other pieces. Placing pieces in the centre therefore greatly facilitates forming groups or connecting areas of the board while also likely obstructing the opponent's ability to do the same. This hypothesis is further supported by the notably poor performance of the *cornerProximity* heuristic, which encourages placing pieces in the corners, areas with the fewest potential contacts on the board.

An interesting avenue for improvement could involve using the *cornerProximity* heuristic in reverse, as a negative heuristic, to actively discourage piece placement in the corners.

5.3.4 Result discussion

The results obtained in the previous section should be interpreted with caution. A crucial step in validating these findings would involve testing the heuristics against a much larger and more diverse panel of agents. This would provide a broader perspective on their effectiveness and robustness.

Additionally, experimenting with alternative configurations could yield valuable insights. For instance, setting an intermediate *playoutValueWeight* could enable a hybrid approach that combines heuristic evaluation with playouts, potentially leading to improved results. Another promising avenue would be to explore the use of these heuristics to enhance the selection stage in MCTS, such as by integrating them into strategies like progressive bias for MCTS [64]. Similarly, these heuristics could be tested within other adapted agents, such as those employing Alpha-Beta pruning, to assess their utility in different decision-making frameworks.

A striking observation of these results is the underperformance of the *topLayerProximity* heuristic, which explicitly leverages 3D insights. Despite its innovative approach, it does not seem to provide an advantage in the current context. Investigating the reasons behind this outcome and exploring the potential to combine this heuristic with others for synergistic improvements could be a compelling direction for future research.

Interestingly, the results suggest that incorporating third-dimensional features into heuristics may not be strictly necessary to achieve promising results in 3D games. Instead, the rules and objectives of the games appear to play a more significant role than the physical layout of the game board. However, these conclusions must be approached with caution, as further validation and analysis are required to confirm their broader applicability.

6

Further Work

In this chapter, we will explore several areas of potential future work that could build upon and extend the findings of this thesis.

6.1 Diversification of 3D Games

In this work, we have primarily focused on Shibumi games with square boards, leveraging their generality to lay the groundwork for the future implementation of other three-dimensional games. A clear direction for further progress is the implementation of additional 3D games. For example, one could consider other Shibumi games not presented in the Shibumi Rule Book [9], such as Sploff [65] or even Shibumi-like games like Akron [66]. These games could likely be implemented with little to no modifications or additions to the existing ludemes.

Another avenue to explore is the design of boards with various shapes, allowing for the adaptation of Shibumi games to different configurations. This would likely involve new piece relations and interactions, bringing new challenges to the design process.

Finally, a significant and intriguing development would be to explore 3D games in different styles, such as Rumis [67], a 3D version of Blokus, which could present unique challenges and opportunities for innovation. However, this would likely require substantial additional work, including remodelling new pieces and boards to accommodate the different gameplay mechanics and spatial dynamics.

6.2 Looking for efficiency

Another obvious direction for further work in this thesis is the study of the effectiveness of the different games that were implemented and examined in this research. In fact, during this thesis, we focused primarily on the functionality and correctness of the various games. However, it would be interesting to assess the efficiency of the different games to determine whether their ludemic representation can be considered 'efficient' (e.g., game display speed, speed of generating potential new moves, etc.).

6.3 Exploring 3D Visualization for Shibumi Games

In all of our Shibumi games, we use a 2D board representation, despite the inherently 3D nature of the game. A promising advance could be to develop a 3D, interactive graphical representation of the board. Such an enhancement would not only highlight the 3D nature of the game but also provide players with a more immersive experience. By visualizing the game in three dimensions, players would gain a clearer understanding of the spatial relationships and dynamics between pieces, helping them to better grasp the complexities of the game.

6.4 Further investigation in AI agents

Another major point of progress would be to produce more advanced work on AI than what we were able to achieve in this study. In fact, in this work, we mainly studied and tested agents that are relatively well-known to the general public, as well as heuristics that are relatively simple and general. Furthermore, we only tested these agents and heuristics on a subset of the games that were available at the time the simulations were carried out. A crucial step would be to conduct simulations firstly on all the games completed by the end of this work, and secondly, to test a much larger range of different agents while developing more complex heuristics, potentially tailored to specific 3D games. This would provide a much clearer understanding of how these agents handle the 3D dynamics of the games we have studied.

7

Conclusion

In this work, we explored the integration of 3D games developed over the past 50–70 years into Ludii, with a primary focus on Shibumi games. We successfully implemented 26 Shibumi games, a significant majority of the games featured in the *Shibumi Rule Book* [9], as well as one additional 3D game into the platform. These implementations are expected to be included in the next Ludii release.

This integration allowed us to incorporate several mechanisms inherently tied to 3D gameplay into the various ludemes used for the ludemic modeling of games in Ludii. These advancements will likely facilitate the implementation of many more games requiring similar features in the future, representing a significant step toward broader integration of 3D games within the platform. What's more, we have rigorously tested all the modifications made to ensure the accuracy, relevance, and conformity of our contributions. These tests were essential to validate the effectiveness of the adjustments and to confirm that they align with the intended objectives.

Finally, we have taken an first step in adapting AI agents using heuristics for 3D games, revealing promising avenues for certain categories of games. Moreover, we have identified several directions for future work that could help extend the generality of Ludii, while also contributing to a deeper understanding of the dynamics and intricacies of 3D board games and how AI agents could tackle them. These insights lay a first groundwork for further development and exploration in this field.

Bibliography

[1] Walter Crist, Anne-Elizabeth Dunn-Vaturi, and Alex de Voogt. *Ancient Egyptians at Play: Board Games Across Borders*. Bloomsbury Egyptology. Bloomsbury Academic, London, 1st edition, 2016.

[2] Georgios N. Yannakakis and Julian Togelius. *Artificial Intelligence and Games*. Springer, 2018.

[3] Michael Genesereth and Michael Thielscher. *General Game Playing*. Morgan & Claypool, 2014.

[4] Robert Canaan, Christoph Salge, Julian Togelius, and Adam Nealen. Leveling the playing field: Fairness in ai versus human game benchmarks. In *Proceedings of the 14th International Conference on the Foundations of Digital Games*, FDG '19, New York, NY, USA, 2019. Association for Computing Machinery.

[5] Michael R. Genesereth, Nathaniel Love, and Barney Pell. General game playing: Overview of the aaai competition. *AI Magazine*, 26(2):62–72, 2005.

[6] Diego Perez-Liebana, Jialin Liu, Ahmed Khalifa, Raluca D. Gaina, Julian Togelius, and Simon M. Lucas. General video game ai: A multitrack framework for evaluating agents, games, and content generation algorithms. *IEEE Transactions on Games*, 11(3):195–214, Sep. 2019.

[7] Tom Schaul. An extensible description language for video games. *IEEE Transactions on Computational Intelligence and AI in Games*, 6(4):325–331, Dec. 2014.

[8] Eric Piette, Dennis JNJ Soemers, Matthew Stephenson, Chiara F Sironi, Mark HM Winands, and Cameron Browne. Ludii–the ludemic general game system. In *ECAI 2020*, pages 411–418. IOS Press, 2020.

[9] Cameron Browne and Néstor Romeral Andrés. *Shibumi Rule Book*. Lulu. com, 2012.

[10] S. Schreiber. Games-base repository. `http://games.ggp.org/base/`„ 2016.

[11] Eric Piette, Frédéric Koriche, Sylvain Lagrue, and Sébastien Tabary. Woodstock: un programme-joueur générique dirigé par les contraintes stochastiques. *Revue d'Intelligence Artificielle*, 2017.

[12] Frédéric Koriche, Sylvain Lagrue, Éric Piette, and Sébastien Tabary. Constraint-based symmetry detection in general game playing. In *IJCAI*, pages 280–287, 2017.

[13] Yngvi Bjornsson and Hilmar Finnsson. Cadiaplayer: A simulation-based general game player. *IEEE Transactions on Computational Intelligence and AI in Games*, 1(1):4–15, 2009.

[14] Jean Méhat and Tristan Cazenave. Ary, a general game playing program. In *Board games studies colloquium*. Citeseer, 2010.

[15] J. Pitrat. Realization of a general game-playing program. In *IFIP Congress (2)*, pages 1570–1574, 1968.

[16] M. Gherrity. *A game-learning machine*. PhD thesis, University of California at San Diego, 1993.

[17] B. Pell. A strategic metagame player for general chess-like games. *Computational Intelligence*, 12, 1996.

[18] S. L. Epstein. Identifying the right reasons: Learning to filter decision makers. In *Proceedings of AAAI'94*, pages 68–71. AAAI Press, 1994.

[19] J. Romein. *Multigame - an environment for distributed game-tree search*. PhD thesis, Vrije Universiteit Amsterdam, 2001.

[20] J. Mallet and M. Lefter. Zillions of games: Unlimited board games & puzzles. `https://www.zillions-of-games.com`, 1998.

[21] Stephan Schiffel and Michael Thielscher. Representing and reasoning about the rules of general games with imperfect information. *Journal of Artificial Intelligence Research*, 49:171–206, 2014.

[22] Michael Thielscher. Gdl-iii: A description language for epistemic general game playing. In *IJCAI*, pages 1276–1282, 2017.

[23] Michael Thielscher. The general game playing description language is universal. In *IJCAI Proceedings-International Joint Conference on Artificial Intelligence*, volume 22, page 1107, 2011.

[24] Yngvi Björnsson and Stephan Schiffel. General game playing. In *Handbook of Digital Games and Entertainment Technologies*, pages 1–23. Springer Singapore, Singapore, 2016.

[25] Hilmar Finnsson and Yngvi Björnsson. Simulation-based approach to general game playing. In *The Twenty-Third AAAI Conference on Artificial Intelligence*, pages 259–264. AAAI Press, 2008.

[26] Hilmar Finnsson and Yngvi Björnsson. Learning simulation control in general game-playing agents. In *The Twenty-Fourth AAAI Conference on Artificial Intelligence*, pages 954–959. AAAI Press, 2010.

[27] Maciej Świechowski, HyunSoo Park, Jacek Mańdziuk, and Kyung-Joong Kim. Recent advances in general game playing. *The Scientific World Journal*, 2015(1):986262, 2015.

[28] Eric Piette, Matthew Stephenson, Dennis JNJ Soemers, and Cameron Browne. An empirical evaluation of two general game systems: Ludii and rbg. In *2019 IEEE Conference on Games (CoG)*, pages 1–4. IEEE, 2019.

[29] Jakub Kowalski, Maksymilian Mika, Jakub Sutowicz, and Marek Szykuła. Regular boardgames. In *Proceedings of the AAAI Conference on Artificial Intelligence*, volume 33, pages 1699–1706, 2019.

[30] Cameron Browne. A class grammar for general games. In *Advances in Computer Games*, volume 10068 of *Lecture Notes in Computer Science*, pages 167–182. Springer, 2016.

[31] Cameron Bolitho Browne. *Automatic generation and evaluation of recombination games*. PhD thesis, Queensland University of Technology, 2008.

[32] Cameron Browne and Frederic Maire. Evolutionary game design. *IEEE Transactions on Computational Intelligence and AI in Games*, 2(1):1–16, 2010.

[33] Cameron Browne. Digital ludeme project, 2018–2023. Accessed: 2024-11-22.

[34] Cameron Browne, Dennis JNJ Soemers, Éric Piette, Matthew Stephenson, Michael Conrad, Walter Crist, Thierry Depaulis, Eddie Duggan, Fred Horn, Steven Kelk, et al. Foundations of digital arch {\ae} oludology. *arXiv preprint arXiv:1905.13516*, 2019.

[35] Dennis JNJ Soemers, Éric Piette, and Cameron Browne. Biasing mcts with features for general games. In *2019 IEEE Congress on Evolutionary Computation (CEC)*, pages 450–457. IEEE, 2019.

[36] Dennis JNJ Soemers, Spyridon Samothrakis, Éric Piette, and Matthew Stephenson. Extracting tactics learned from self-play in general games. *Information Sciences*, 624:277–298, 2023.

[37] Graham Todd, Alexander Padula, Matthew Stephenson, Éric Piette, Dennis J. N. J. Soemers, and Julian Togelius. Gavel: Generating games via evolution and language models. In *Advances in Neural Information Processing Systems 37 (NeurIPS 2024)*, 2024. Accepted.

[38] Matthew Stephenson, Dennis JNJ Soemers, Éric Piette, and Cameron Browne. Measuring board game distance. In *International Conference on Computers and Games*, pages 121–130. Springer, 2022.

[39] Walter Crist, Éric Piette, DJNJ Soemers, Matthew Stephenson, and Cameron Browne. Computational approaches for recognising and reconstructing ancient games: The case of ludus latrunculorum. *The Archaeology of Play: Material Approaches to Games and Gaming in*, 2023.

[40] Éric Piette, Cameron Browne, and Dennis JNJ Soemers. Ludii game logic guide. *arXiv preprint arXiv:2101.02120*, 2021.

[41] Cameron Browne. Everything's a ludeme well, almost everything. In *XXIII BOARD GAME STUDIES COLLOQUIUM-The Evolutions of Board Games*, 2021.

[42] David Parlett. What'sa ludeme. *Game & Puzzle Design*, 2(2):81–84, 2016.

[43] Cameron Browne, Matthew Stephenson, Éric Piette, and Dennis JNJ Soemers. A practical introduction to the ludii general game system. In *Advances in Computer Games: 16th International Conference, ACG 2019, Macao, China, August 11–13, 2019, Revised Selected Papers 16*, pages 167–179, 2020.

[44] Cameron Browne, Éric Piette, Matthew Stephenson, and Dennis Soemers. Ludii general game system for modeling, analyzing, and designing board games. In *Encyclopedia of Computer Graphics and Games*. Springer, Cham, 2023.

[45] Walter Crist, Matthew Stephenson, Éric Piette, and Cameron Browne. The ludii games database: A resource for computational and cultural research on traditional board games. *Digital Humanities Quarterly*, 18(4), 2024.

[46] Matthew Stephenson, Eric Piette, Dennis JNJ Soemers, and Cameron Browne. An overview of the ludii general game system. In *2019 IEEE Conference on Games (CoG)*, pages 1–2. IEEE, 2019.

[47] Cameron B Browne, Edward Powley, Daniel Whitehouse, Simon M Lucas, Peter I Cowling, Philipp Rohlfshagen, Stephen Tavener, Diego Perez, Spyridon Samothrakis, and Simon Colton. A survey of monte carlo tree search methods. *IEEE Transactions on Computational Intelligence and AI in games*, 4(1):1–43, 2012.

[48] Matthew Stephenson, Dennis JNJ Soemers, Éric Piette, and Cameron Browne. General game heuristic prediction based on ludeme descriptions. In *2021 IEEE Conference on Games (CoG)*, pages 1–4. IEEE, 2021.

[49] Dennis JNJ Soemers, Éric Piette, Matthew Stephenson, and Cameron Browne. The ludii game description language is universal. In *2024 IEEE Conference on Games (CoG)*, pages 1–8. IEEE, 2024.

[50] Dennis JNJ Soemers, Jakub Kowalski, Éric Piette, Achille Morenville, and Walter Crist. Gametable working group 1 meeting report on search, planning, learning, and explainability. *ICGA Journal*, pages 1–8, 2024.

[51] Achille Morenville, Éric Piette, and UCLouvain ICTEAM. Vers une approche polyvalente pour les jeux à information imparfaite sans connaissance de domaine. In *Plate-Forme d'Intelligence Artificielle-Rencontres des Jeunes Chercheurs en Intelligence Artificielle*, 2024.

[52] Achille Morenville and Éric Piette. Belief stochastic game: A model for imperfect-information games with known positions. In *Computer and Games*, 2024.

[53] Cambolbro. Margo basics 41, 2024.

[54] Cameron Browne, Éric Piette, Matthew Stephenson, and Dennis JNJ Soemers. General board geometry. In *Advances in Computer Games*, pages 235–246. Springer, 2021.

[55] Kent Beck. *Test driven development: By example*. Addison-Wesley Professional, 2022.

[56] Donald E Knuth and Ronald W Moore. An analysis of alpha-beta pruning. *Artificial intelligence*, 6(4):293–326, 1975.

[57] Guillaume Chaslot, Sander Bakkes, Istvan Szita, and Pieter Spronck. Monte-carlo tree search: A new framework for game ai. In *Proceedings of the AAAI Conference on Artificial Intelligence and Interactive Digital Entertainment*, volume 4, pages 216–217, 2008.

[58] Frederik Christiaan Schadd. *Monte-Carlo search techniques in the modern board game Thurn and Taxis*. PhD thesis, MS thesis, Maastricht Univ., Netherlands, 2009.

[59] Guillaume M Jb Chaslot, Mark HM Winands, H Jaap van den Herik, Jos WHM Uiterwijk, and Bruno Bouzy. Progressive strategies for monte-carlo tree search. *New Mathematics and Natural Computation*, 4(03):343–357, 2008.

[60] P Auer. Finite-time analysis of the multiarmed bandit problem, 2002.

[61] Levente Kocsis, Csaba Szepesvári, and Jan Willemson. Improved monte-carlo search. *Univ. Tartu, Estonia, Tech. Rep*, 1:1–22, 2006.

[62] Tristan Cazenave. Generalized rapid action value estimation. In *24th International conference on artificial intelligence*, pages 754–760, 2015.

[63] CÉCI. Clusters at céci. `https://www.ceci-hpc.be/clusters.html`, n.d. Accessed: 2025-01-03.

[64] Guillaume Maurice Jean-Bernard Chaslot Chaslot. Monte-carlo tree search. 2010.

[65] Cambolbro. Spoff, 2025. Accessed: 2025-01-04.

[66] Cambolbro. Akron, 2025. Accessed: 2025-01-04.

[67] DadsGamingAddiction. Rumis, 2025. Accessed: 2025-01-04.

Appendix

A: GDL and RBG implémentation of Tic Tac Toe

```
(role xplayer)  (role oplayer)

(index 1) (index 2) (index 3)
(<= (base (cell ?x ?y b)) (index ?x) (index ?y))
(<= (base (cell ?x ?y x)) (index ?x) (index ?y))
(<= (base (cell ?x ?y o)) (index ?x) (index ?y))
(<= (base (control ?p)) (role ?p))
(<= (input ?p (mark ?x ?y)) (index ?x) (index ?y) (role ?p))
(<= (input ?p noop) (role ?p))

(init (cell 1 1 b))  (init (cell 1 2 b))  (init (cell 1 3 b))
(init (cell 2 1 b))  (init (cell 2 2 b))  (init (cell 2 3 b))  (init (cell 3 1 b))
  (init (cell 3 2 b))  (init (cell 3 3 b))
(init (control xplayer))

(<= (next (cell ?m ?n x))(does xplayer (mark ?m ?n)) (true (cell ?m ?n b)))
(<= (next (cell ?m ?n o)) (does oplayer (mark ?m ?n)) (true (cell ?m ?n b)))
(<= (next (cell ?m ?n ?w)) (true (cell ?m ?n ?w)) (distinct ?w b))
(<= (next (cell ?m ?n b)) (does ?w (mark ?j ?k)) (true (cell ?m ?n b)) (or (distinct
  ?m ?j) (distinct ?n ?k)))
(<= (next (control xplayer)) (true (control oplayer)))
(<= (next (control oplayer)) (true (control xplayer)))
(<= (row ?m ?x) (true (cell ?m 1 ?x)) (true (cell ?m 2 ?x)) (true (cell ?m 3 ?x)))
(<= (column ?n ?x) (true (cell 1 ?n ?x)) (true (cell 2 ?n ?x)) (true (cell 3 ?n ?x)))
(<= (diagonal ?x) (true (cell 1 1 ?x)) (true (cell 2 2 ?x)) (true (cell 3 3 ?x)))
(<= (diagonal ?x) (true (cell 1 3 ?x)) (true (cell 2 2 ?x)) (true (cell 3 1 ?x)))
(<= (line ?x) (row ?m ?x))
(<= (line ?x) (column ?m ?x))
(<= (line ?x) (diagonal ?x))
(<= open (true (cell ?m ?n b)))

(<= (legal ?w (mark ?x ?y)) (true (cell ?x ?y b)) (true (control ?w)))
(<= (legal xplayer noop) (true (control oplayer)))
(<= (legal oplayer noop) (true (control xplayer)))

(<= (goal xplayer 100) (line x))
(<= (goal xplayer 50) (not (line x)) (not (line o)) (not open))
(<= (goal xplayer 0) (line o))
(<= (goal oplayer 100) (line o))
(<= (goal oplayer 50) (not (line x)) (not (line o)) (not open))
(<= (goal oplayer 0) (line x))

(<= terminal (line x))
(<= terminal (line o))
(<= terminal (not open))
```

Figure A.1: GDL Implementation of Tic Tac Toe
Source: https://games.ggp.org/base/games/ticTacToe/ticTacToe.kif

```
#players = xplayer(100), oplayer(100)
#pieces = e, x, o
#variables =
#board = rectangle(up,down,left,right,
    [e, e, e]
    [e, e, e]
    [e, e, e])

#anySquare = ((up + down)(left + right))

#winAtDir(dir; oppDir; piece) = (dir {piece} (dir + (oppDir)^2) {piece} + (oppDir
  {piece})^2)
#winAt(piece) = (
      winAtDir(up left; down right; piece)
    + winAtDir(up; down; piece)
    + winAtDir(up right; down left; piece)
    + winAtDir(left; right; piece)
  )

#turn(me; opp) =
  ->me~player anySquare {e} ->> [me]
  (
      {! winAt(me)}
    + {? winAt(me)} [$ me~player=100] [$ opp~player=0] ->> {}
  )

#rules = [$ xplayer=50] [$ oplayer=50] (
    turn(x; o)
    turn(o; x)
  )
```

Figure A.2: RBG Implementation of Tic Tac Toe
Source: https://github.com/marekesz/rbgGames/TicTacToe

B: Ludemic implementation of Spava in Ludii

```
(game "Spava"
    (players 2)
    (equipment {
        (board (square 4 pyramidal:True) use:Vertex)
        (piece "Ball" Each)
    })
    (rules
        (play
            (move Add
                (to (sites Empty)
                    if:(is Flat)
                )
            )
        )
        (end {
            (if
                (is Line (- (count Rows) (layer of:(last To))) SameLayer)
                (result Mover Win)
            )
            (if
                (is Line (- (- (count Rows) (layer of:(last To))) 1) SameLayer)
                (result Mover Loss)
            )
        })
    )
)
```

Figure B.3: Ludemic implementation of Spava in Ludii
Source: https://github.com/Ludeme/Ludii

C: All implemented Shibumi games

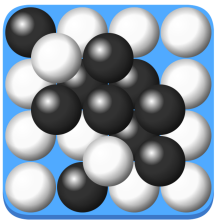

Figure C.4: Spline+ Game won by White

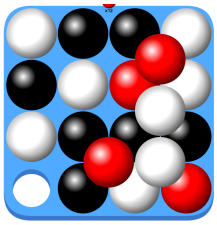

Figure C.5: Splice Game won by White

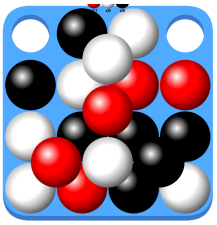

Figure C.6: Spree Game won by Black

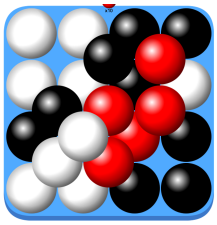

Figure C.7: Alternative Spava Game won by White

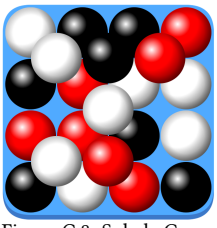

Figure C.8: Splade Game won by White

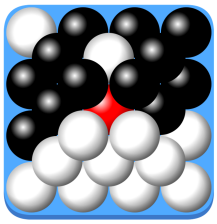

Figure C.9: Sparro Game won 7-3 by White

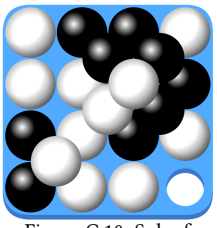

Figure C.10: Sploof Game won by White

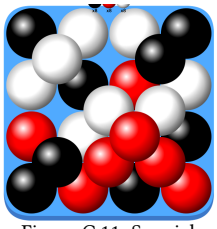

Figure C.11: Spaniel Game won by Red

Figure C.12: Span Game won by White

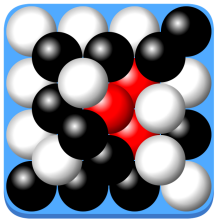

Figure C.13: Sponnect Game won by Black

Figure C.14: Spight Game won by White

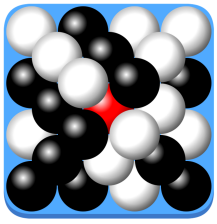

Figure C.15: Spice Game won 8-6 by White

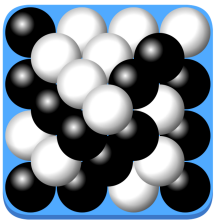
Figure C.16: Span Game won by Black

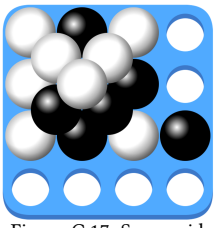
Figure C.17: Spyramid Game won by White

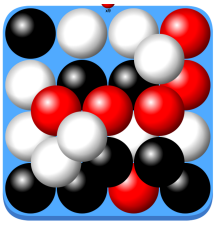
Figure C.18: Spire Game won by White

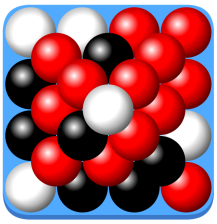
Figure C.19: Spinimax Game won by White

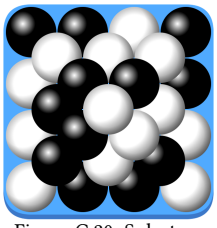
Figure C.20: Splastwo Game won 3-2 by White

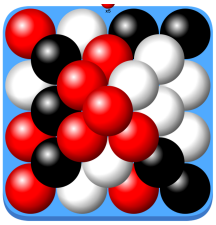
Figure C.21: Sprite Game won 3-0 by Black

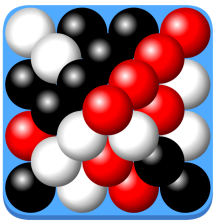
Figure C.22: Spao Game won 25 to 16 and 18 by Red

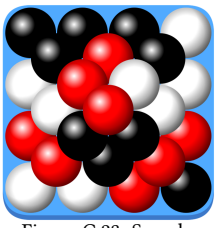
Figure C.23: Speedo Game won 18 to 16 and 15 by Red

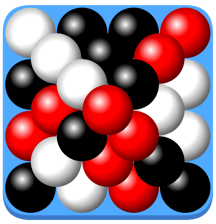
Figure C.24: Spirit Game tied with all player having 10 point

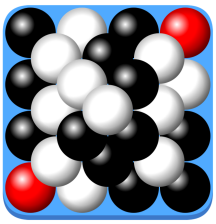
Figure C.25: Sprite Game won 3-2 by White

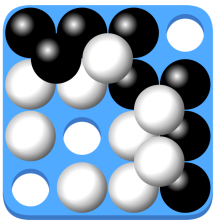
Figure C.26: Spargo Game won 9-8 by White

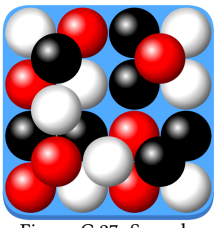
Figure C.27: Spuzzle Game ended with score 22

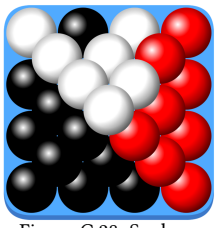
Figure C.28: Spalone game successfully solved

D: MODIFIED SUPERLUDEMES ASSOCIATED GRAPHS

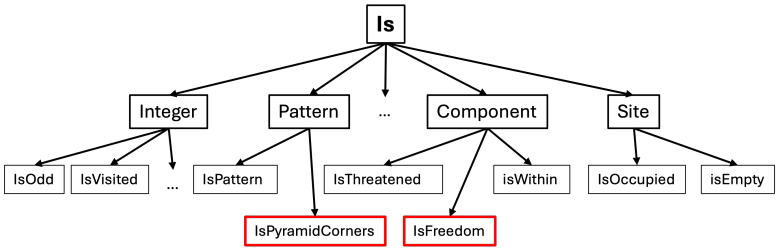

Figure D.29: Simplified graph associated with superludeme Is (missing sub-elements are indicated by three dots and red counters highlight the elements created during this work)

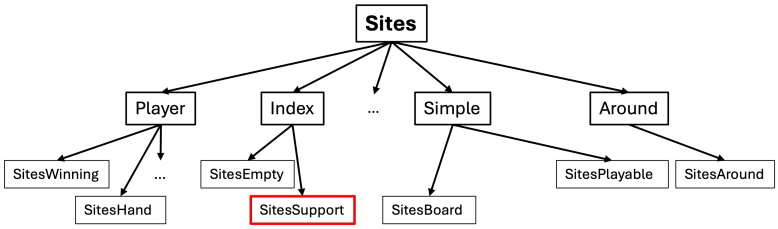

Figure D.30: Simplified graph associated with superludeme Sites (missing sub-elements are indicated by three dots and red counters highlight the elements created during this work)

UNIVERSITÉ CATHOLIQUE DE LOUVAIN
École polytechnique de Louvain
Rue Archimède, 1 bte L6.11.01, 1348 Louvain-la-Neuve, Belgique | **www.uclouvain.be/epl**