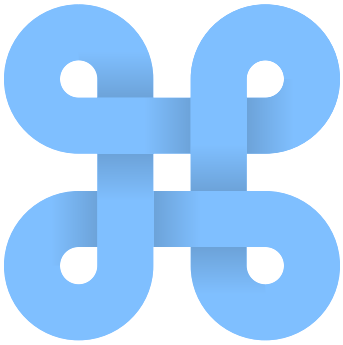


Ludii User Guide

Pre-release (August 13, 2019)

Dennis J. N. J. Soemers,
Éric Piette, Matthew Stephenson and Cameron Browne

Department of Data Science and Knowledge Engineering (DKE)
Maastricht University
Maastricht, the Netherlands

August 12, 2019

Introduction

This user guide describes the basic functionality of Ludii – a "general game system" that can be used to play a wide variety of games. Ludii is a program developed for the ERC-funded Digital Ludeme Project, in which mathematical and computational approaches are used to study how games were played, and spread, throughout history. Primary features of Ludii include:

- A large library of implemented games, which can be played by any combination of human and computer players, and displayed graphically in the application.
- A new *ludeme*-based game description language, which allows for new games to be modelled in more succinct and human-friendly formats than prior game description languages, and run more efficiently.
- Implementations of common game-playing algorithms from the Artificial Intelligence literature, and the ability to write and import custom agents in the application.

Ludii primarily contains board game, dice games, card games, puzzles, and other abstract games, but also contains a single real-time game as a proof of concept.

This user guide was written for the "pre-release" version of Ludii – a version intended for early adopters, who are welcome to provide feedback and suggestions on the application! The first full release of Ludii is planned for January 1, 2020.

To remain up-to-date with further developments of Ludii and the Digital Ludeme Project, keep an eye on the websites and twitter pages for each of them:

- **Ludii website**: http://ludii.games/
- **Ludii Twitter**: https://twitter.com/ludiigames
- **Digital Ludeme Project website**: http://www.ludeme.eu/
- **Digital Ludeme Project Twitter**: https://twitter.com/archaeoludology/

Please contact the authors with any comments or corrections at: ludii.games@gmail.com

Contents

1 Installation and Running

1.1 Prerequisites

Ludii requires Java version 8 or higher to be installed on your computer. Java can be downloaded from: https://www.java.com/download/. Ludii should run correctly on any major operating system. It has been verified to run correctly on the following operating systems:

- Windows 10.
- OS X El Capitan 10.11.6, OS X Mojave 10.14.3.
- Ubuntu 16.04, Ubuntu 18.04, Ubuntu 19.04.

1.2 Installation

The latest version of Ludii can be downloaded from: http://ludii.games/downloads.html. The downloaded file will be named `Ludii-x.y.z.jar`, where the `x`, `y`, and `z` denote major, minor and patch versions, respectively.

Additional installation steps after downloading the file are not required. Because Ludii may write additional files in the directory containing the `.jar` file, we recommend placing it in a directory that is otherwise unused (for example: `C:/Program Files/Ludii/Ludii-x.y.z.jar`).

1.3 Running

The easiest way to launch Ludii is to double-click the downloaded `.jar` file. Alternatively, it may be launch by navigating to the directory containing the `.jar` file in a command prompt, and entering:

```
java -jar Ludii-x.y.z.jar
```

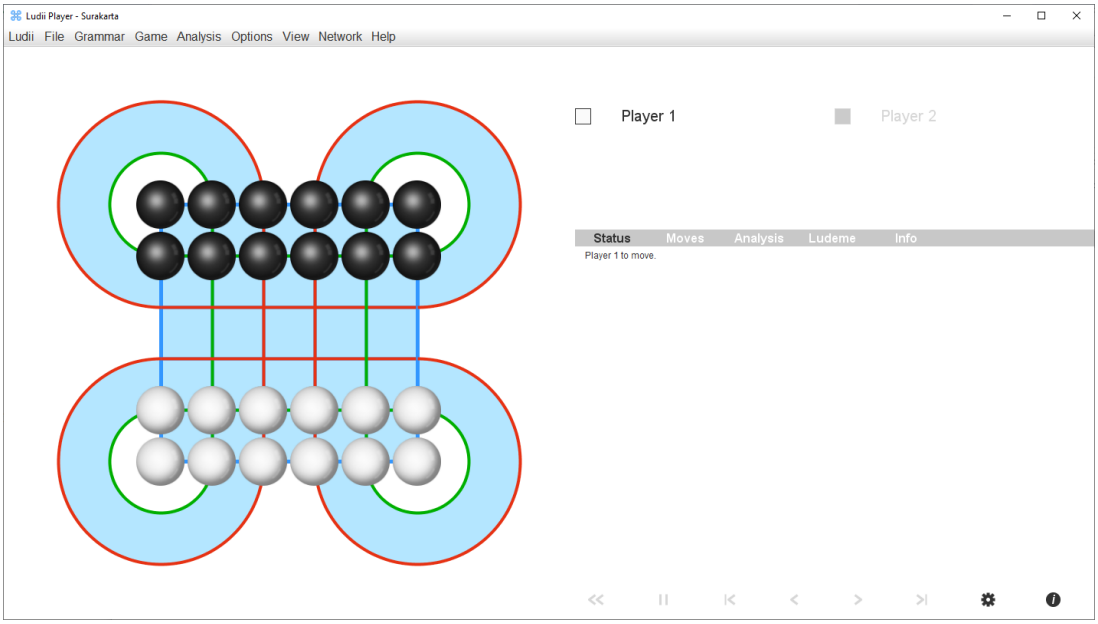


Figure 1.1: The initial screen presented by Ludii, with the game *Surakarta*.

When launching the application for the first time, it will load the game *Surakarta* by default, and present the screen depicted in Figure 1.1.

2
Basic Usage

Basic usage of Ludii primarily involves loading and playing any of the games included in its default library of games.

2.1 Overview of the User Interface

The graphical user interface (GUI) of Ludii can be split up in four main areas, in addition to its menubar.

2.1.1 Main Areas

The four main areas are depicted in coloured (and numbered) rectangles in Figure 2.1.

Playing Area The *playing area* (area 1 in Figure 2.1) is the main area that players interact with when playing games. In most games, it depicts a board with any placed pieces. In games that do not use a board, it may just display pieces that have been placed on the "table". In most games, human players can select their moves by clicking and/or dragging pieces in this area.

Players' Area The *players' area* (area 2 in Figure 2.1) shows basic information about the players, such as their names and colours. Clicking any player's name opens a dialog that can be used to adjust various preferences (see sections 2.1.3 to 2.1.5). In some games it may also contain the outcomes of dice rolls, or depict "hands" containing pieces that players may be able to drag to the playing area. An example of a game where this area contains dice (*Backgammon*) is depicted in Figure 2.2. An example of a game where this area contains pieces that may be dragged onto the board (*Three Men's Morris*) is depicted in Figure 2.3.

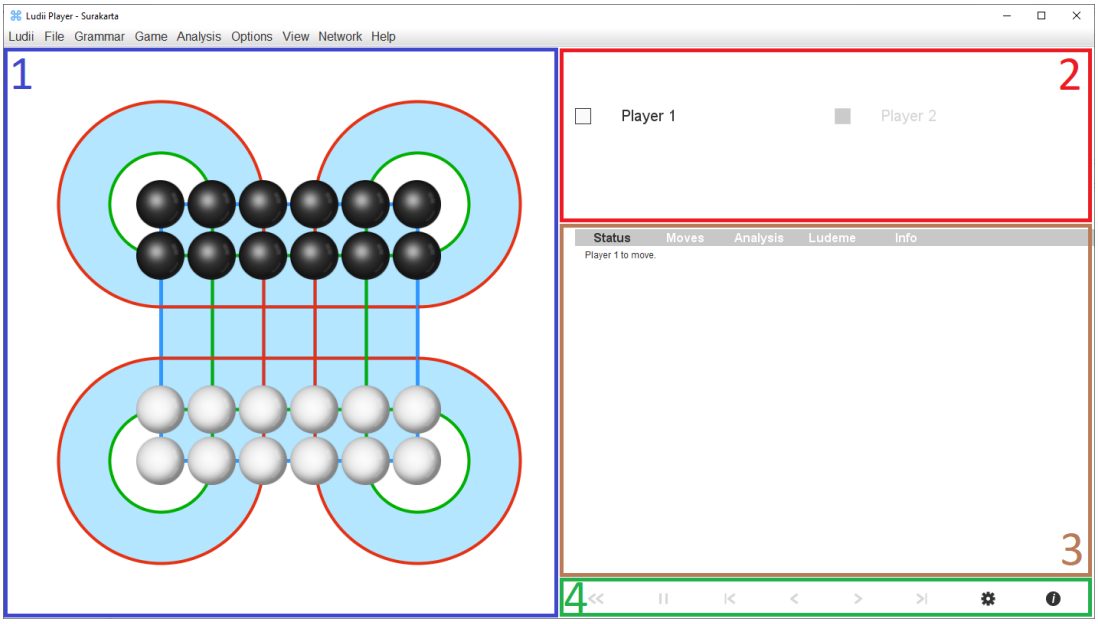


Figure 2.1: The graphical user interface (GUI) of Ludii. Coloured rectangles mark the playing area (1), the players' area (2), the panels area (3), and the buttons area (4).

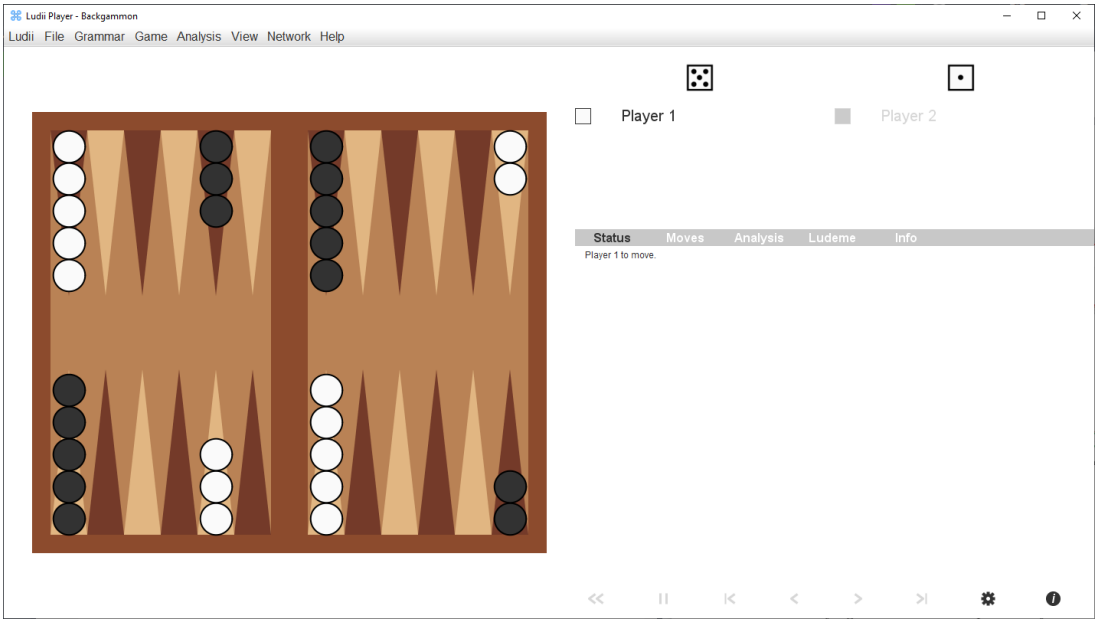


Figure 2.2: The game *Backgammon* in Ludii. The players' area shows two dice that have been rolled.

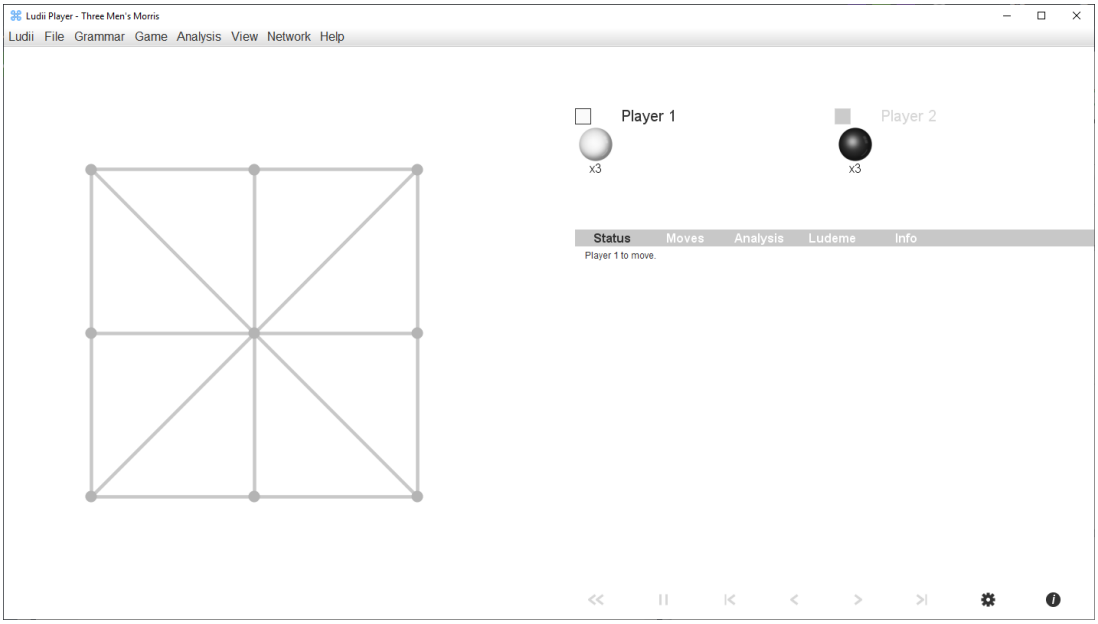


Figure 2.3: The game *Three Men's Morris* in Ludii. The players' area shows pieces that can be dragged onto the board by their owners.

Panels Area The *panels area* (area 3 in Figure 2.1) contains a panel in which Ludii displays information. The panel has the following tabs:

- **Status**: this tab displays the status of the current game being played. This includes, for example, showing which player is the current player to move, or showing the winner of a game when it is finished.
- **Moves**: this tab displays the sequence of moves that has been applied in the current game. This information is primarily intended for developers of Ludii or third-party AI players.
- **Analysis**: this tab sometimes displays information about AI players.
- **Ludeme**: this tab displays the game description of the currently loaded game, in its *ludeme*-based format (see Chapter 3).
- **Info**: this tab displays a short English description of the currently loaded game's rules.

Buttons Area The *buttons area* (area 4 in Figure 2.1) may, depending on the context, display any of the following buttons:

- ⏪ : if there is a current game in progress, this button can be used to reset to an initial game state. The application will not remember any progress from the previous game.
- ▶ : if the current player to move has been set to be an AI player (see Subsection 2.1.4), this button can make it start playing. Any subsequent AI movers will automatically continue playing.
- ⏸ : if any AI players are currently playing, this button can be used to pause them.

- |< : if there is a current game in progress, this button winds back to the start of the game. The application will remember the moves that were selected thus far, and will allow re-applying them.
- < : if there is a current game in progress, this button takes a single step back (e.g. to the previous turn in a turn-based game).
- > : this button takes a step forward (e.g. re-applies one move) if we previously stepped backwards through a game in progress.
- >| : this button re-applies any moves that we previously stepped backwards through in a game in progress.
- : this button opens the same dialog with settings that can also be opened by clicking any player name (see sections 2.1.3 to 2.1.5).
- : this button opens the "About" dialog of Ludii, which provides information about the version number, authors, acknowledgements, etc.

2.1.2 Menu Bar Options

The menu bar at the top of Ludii's user interface contains various menus with additional options. This subsection provides short explanations for all options that can be found in these menus. Some of the more advanced options are explained in more detail in subsequent sections and chapters. Most of these options can also be accessed using keyboard shortcuts, which are listed in Appendix A.

Ludii The Ludii menu has two simple options:

- **About**: opens Ludii's "About" dialog.
- **Quit**: closes Ludii.

File The File menu contains options for loading games, and saving and loading games in progress:

- **Load Game**: opens Ludii's game loader (depicted in Figure 2.4), which allows for any of Ludii's built-in games to be loaded. The games are sorted in different categories, very much like a file browser with files sorted in directories.
- **Load Game from File**: opens a standard file browser, which can be used to navigate to any directory on your computer and load a game description from a `.lud` file. This may be used to load custom games that were not originally included in the Ludii download (see Chapter 3). Note that, for security reasons, we recommend only loading files provided by trusted sources!
- **Load Trial**: opens a file browser that can be used to load a previously saved "trial", which is a record of a game in progress. This may be used to resume playing a game that was previously saved.
- **Save Trial**: opens a file browser that can be used to save the progress of the current game to a file, allowing it to be loaded and resumed in the future.

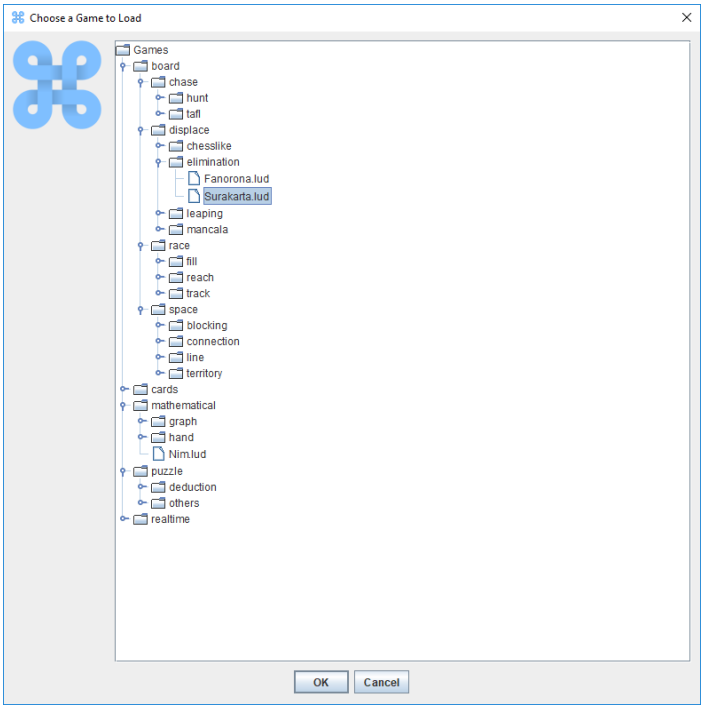


Figure 2.4: Ludii's Game Loader.

Grammar The Grammar menu contains options that will be primarily of interest to users interested in modelling new games for Ludii (see Chapter 3):

- **Generate Grammar**: prints the grammar that defines the language used by Ludii to model games in an extended Backus-Naur form (EBNF)-like notation. The output is written to the standard output stream of the Ludii process, and will typically only be visible if the application was launched using a command line.

Game The Game menu contains some options for interaction with the current game:

- **Restart**: restarts the current game.
- **Random Move**: randomly selects one random move from all moves available in the current game state, and applies it.
- **Random Playout**: starting from the current game state, rolls out all the way to a terminal game state by selecting a random sequence of legal moves.
- **Time Random Playouts**: continuously runs full, random playouts in the selected game, over a period of fourty seconds. The average number of full playouts per second, measured over the last thirty[1] seconds, are reported in the Status tab of the panels area. This gives a rough[2] indication of the efficiency of any game's implementation in Ludii.

[1]The first ten seconds are used to "warm up" the Java Virtual Machine.

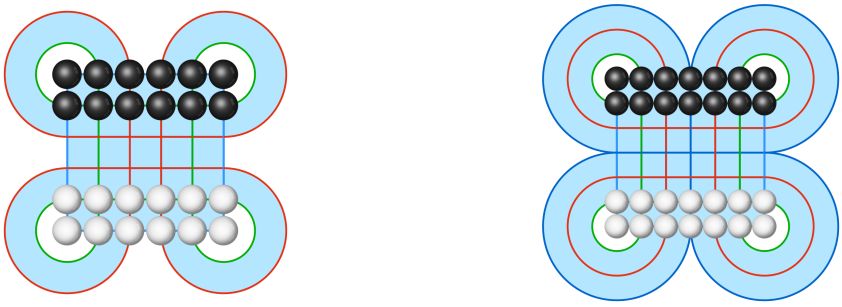
[2]To obtain more accurate numbers, for instance for academic publications, we recommend timing playouts without a GUI, and measuring over a longer period of time.

- **Game Screenshot**: takes a screenshot of the playing area, and saves it to a new `.png` file in the directory that contains Ludii's `.jar` file (or the current working directory if Ludii was launched from a command line).
- **Auto From Moves**: if this option is toggled on, Ludii will automatically move a piece to a location that the user clicks on if there is only a single move possible towards the clicked location. In other cases (e.g. if there are multiple possible moves to the same location), the user will still be required to drag the piece to be moved, or click on the location to move from before clicking on the location to move to.
- **Auto To Moves**: if this option is toggled on, Ludii will automatically move a piece that the user clicks on to a new location if there is only a single move possible from the location that was clicked on. In other cases (e.g. if there are multiple possible moves from the same location), the user will still be required to drag the piece to be moved, or click on the location to move from before clicking on the location to move to.

Analysis The Analysis menu contains an option that will be primarily of interest for AI researchers and developers of new AI players for Ludii:

- **Evaluate AI vs. AI**: if all players for the current game are set to an AI player (see Subsection 2.1.4), this option will start a sequence of 100 games between the AI players to evaluate their relative performance levels in the selected game. After every finished game, updated statistics on the evaluation so far are printed in the "Analysis" tab of the panels area. The evaluation can be interrupted at any time by clicking the pause button. For a fair evaluation, Ludii will automatically rotate through all possible assignments of selected AI players to player number (or colours).

Options The Options menu is not always visible, but only if the currently selected game has a set of options in its *ludeme*-based game description. The set of options that are available also varies from game to game. The most common options are to select different board sizes or rulesets. For example, in the implementation of *Surakarta* included in Ludii, there is an option to add extra lines through the middle of the board, with an extra loop around every corner. This also changes the initial number of pieces for each player from twelve to fourteen. This is depicted in Figure 2.5.



(a) *Surakarta* with the official board. (b) *Surakarta* variant with modified board.

Figure 2.5: Ludii has two options available in *Surakarta*.

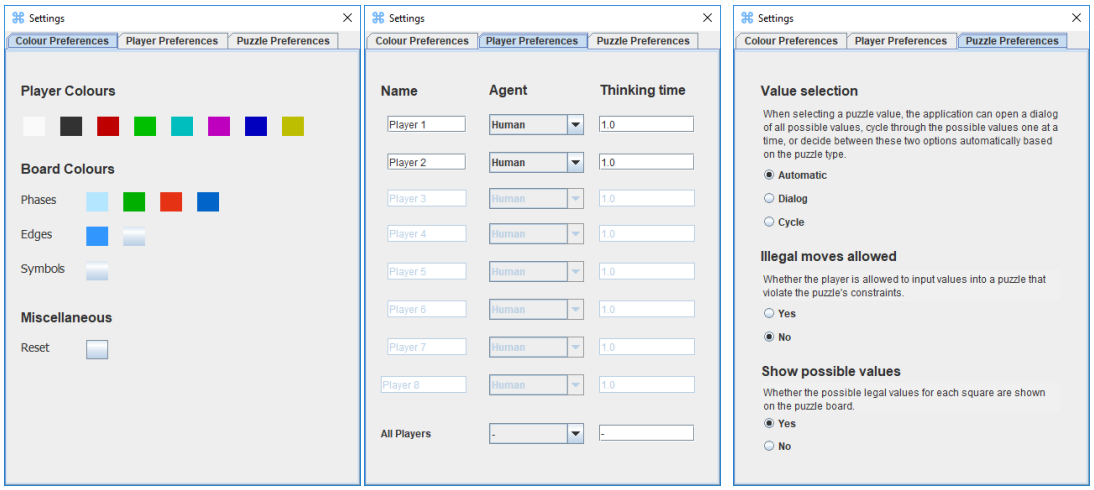
View The View menu provides a variety of settings to display extra information in Ludii's user interface:

- **Show Board**: if turned off, Ludii will not draw boards for any games.
- **Show Graph**: if turned on, Ludii draws the underlying graph representation of a board, used internally by Ludii to determine adjacencies of cells, on top of the board.
- **Show Axes**: if turned on, Ludii adds extra labels around boards which may be useful to refer to specific locations when playing with other humans. For instance, in *Chess* it labels the rows with numbers 1 through 8, and columns with letters A through H.
- **Show Possible Moves**: if turned on, Ludii draws a blue dot on every location that may be clicked for a legal move in the current game state. If a move involves two locations (for instance to move a piece from one location to another in a game like *Chess*), Ludii will draw a red dot on every valid destination after a starting location has been selected by clicking on it. Note that valid locations may not only be located in the playing area, but sometimes also in the players' area (see, for instance, *Three Men's Morris*).
- **Show Last Move**: if turned on, Ludii highlights the previous move made in a game with a yellow circle or arrow.
- **Show Cell Indices**: if turned on, Ludii draws a unique number in every location in the current game. This is primarily intended for developers of Ludii or third-party games or AI players for Ludii.
- **Show Edge Indices**: if turned on, Ludii draws a unique number on top of every edge of a cell in the current game. This is primarily intended for developers of Ludii or third-party games or AI players for Ludii.
- **Show Face Indices**: if turned on, Ludii draws a unique number in every face of the underlying graph of the current game's board. This is primarily intended for developers of Ludii or third-party games or AI players for Ludii.
- **Show Coordinates**: if turned on, Ludii draws labels – constructed from the labels shown around boards by the "show axes" option – in every cell.
- **Show AI Distribution**: if turned on, depending on which AI players are currently active, Ludii may visualise their "thought process".
- The **Colour Preferences**, **Player Preferences** and **Puzzle Preferences** dialogs are discussed in sections 2.1.3 to 2.1.5.
- **Show Tracks**: in games with tracks (such as many racing games), this can be used to visualise the tracks that different players move along.

Network The Network menu allows for users to connect to each other and play Ludii games over a network. A detailed explanation of this can be found in Section 2.4.

2.1.3 Colour Preferences

Most of the colours that appear in various places in Ludii can be customised using the Colour Preferences dialog (see Figure 2.6a). There are options to change the colours of players' pieces, and different aspects of the boards. The exact options present depend on the selected game. Some games, such as *Hex*, have custom player colours specified in their game descriptions, but these may still be overridden by user preferences.



(a) Colour preferences. (b) Player preferences. (c) Puzzle preferences.

Figure 2.6: Different dialogs for various preferences in Ludii.

2.1.4 Player Preferences

The primary purpose of the Player Preferences dialog (see Figure 2.6b) is to assign players to be controlled by humans or different types of AI players. By default, Ludii expects every player to be human-controlled, but any player can be switched to AI-controlled by selecting an AI player using the drop-down menus in the middle column. Additionally, the dialog allows for player names to be customised, and may be used to change the maximum amount of "thinking time" that AI-controlled players are allowed to use (in seconds). By default, AIs use 1 second (or less) per move. We briefly summarise what players may expect in terms of playing strength from the different AI options in this section. More detailed and technical explanations of the different AI options can be found in Appendix B.

- **Random**: a simple computer player that randomly selects moves.
- **Monte Carlo (flat)**: a weak AI which should be easily beaten by humans in almost every game.
- **UCT**: a standard AI technique, for which the playing strength greatly depends on game complexity. In simple games like *Tic-Tac-Toe* it should easily be capable of perfect play with the default setting of 1 second per move, but in complex games like *Chess* it will still be easily beaten.
- **MC-GRAVE**: in most games, this player is expected to outperform UCT, but it is possible that it may be weaker in some games.
- **Biased MCTS**: in most of the games that it is able to play, this is expected to be the strongest built-in AI player of Ludii (in some it may be outperformed by MC-GRAVE). However, it only supports a limited selection of games in the current version of Ludii, and will be automatically replaced by UCT in other games. When this happens, this will be printed to the Status tab.
- **From JAR**: this option may be used to load third-party AIs from `.jar` files (see Chapter 4).

2.1.5 Puzzle Preferences

Players in Ludii typically interact differently with most (deductive) puzzles than they do with other games. These deductive puzzles typically consist of locations, each of which may be assigned a variety of values. For example, in *Sudoku*, every cell may be assigned a value ranging from 1 to 9. The Puzzle Preferences dialog has a few options for how users interact with the Ludii user interface in these puzzles:

- **Value selection**: When selecting a puzzle value, the application can open a dialog of all possible values, cycle through the possible values one at a time, or decide between these two options automatically based on the puzzle type.
- **Illegal Moves allowed**: Whether the player is allowed to input values into a puzzle that violate the puzzle's constraints.
- **Show possible values**: Whether the possible legal values for each location are shown on the puzzle board.

2.2 Playing Games with Human Players

By default, all players in Ludii are set to human-controlled. This means that humans who wish to play together on a shared screen can directly start playing after loading any game in Ludii. If any players were previously set to be AI-controlled, they may be switched back to Human in the Player Preferences dialog. If AIs are paused, human input can also be used to select moves for players that have been set to be AI-controlled without first switching them back to Human.

The controls for most games should be intuitive; the mouse is typically used to place pieces by clicking in locations (in games like *Hex*, *Reversi*, etc.), or used to move pieces by dragging them or by clicking consecutively on the piece and the destination. It may be helpful to use the "Show Possible Moves" option from the View menu to visualise possible moves – in particular when trying new games for which you are not yet familiar with the rules!

2.2.1 Games with Hidden Information or Simultaneous Moves

Ludii contains some game with hidden information (like *Stratego*), and games in which players are expected to select their moves simultaneously (like *Rock-paper-scissors*). Ludii allows for them to be played on a single screen, but this may not be ideal in practice because players may get access to information they should not have. Playing such games over the network circumvents these issues.

2.2.2 Real-time Games

The pre-release of Ludii includes a single real-time game as a proof of concept; a real-time variant of *Tron*. Unlike other games, these games are played using the keyboard instead of the mouse. Player 1 may change the direction of its piece using the WASD keys, and player 2 can do the same using the IJKL keys.

2.3 Playing Games with Computer Players

After setting one or more players in the Player Preferences to be AI-controlled, Ludii can be used to play against computer players (or even just watch as multiple computer players play against each other). If the Play button (▶) is visible in the bottom of the user interface, it can be used to make the AI player start playing. Any subsequent AI-controlled players will automatically continue playing when it's their turn to move, unless they are paused using the Pause button (⏸). When playing against AIs, it may sometimes be useful to toggle on the "Show Last Move" option in the View menu, to avoid confusion about which move the previous AI-controlled player last made.

2.3.1 Games with Hidden Information

When AI-controlled players play games with hidden information, currently they will still have access to all information (even the information that should be hidden). The two simplest AI players (*Random* and *Monte Carlo (flat)*) are not able to abuse this, but all others are – and will likely easily beat most human players as a result. Ensuring that AI players only receive the information that they should have is still work in progress.[3]

2.4 Playing Games over a Network

The current version of Ludii supports several separate applications playing games on different machines within the same local network. Each machine will only be able to control the pieces of at least one player, and all relevant hidden information for that player will not be visible.

To enable network play, select the Network/Connect option within the application on all machines. This will open a Network Dialog. Closing this Network Dialog will close the network connection, so be sure to keep it open while the network is in use.

One device must operate as the host. Each machine's IP address should be entered automatically into the "Host IP" box if it can be detected, otherwise it will need to be entered manually. The host machine can also act as a spectator by selecting the appropriate check box, in which case the host will not take control of any player. If the host does not act as a spectator then the host machine will take control of player one. The host can be started by clicking the "Host Start" button. If the host has been set up correctly, then a message will display showing the hosts IP address and port number.

All other client machines can then connect to the host using the host's IP and port number. Type these values into the required boxes, and then click the "Connect" button. If the connection was successful a confirmation message will appear on the host machine. Each client machine will be allocated a player ID by the host, based on the order in which they were connected.

Any valid moves made on any of the client or host machines will then be sent to all other machines. If there is some problem with the network or the machines get out of sync, then received actions may not be valid. If this happens, the host will need to restart the game to re-sync the clients with the host's game state. Hosts and Clients can send messages to each other using the "Chat" section. All client machines will automatically load and restart games each time the host does.

[3] More precisely, we are still investigating what the most sensible way to implement a forward model in this category of games would be.

Playing over the network is not yet supported for the proof-of-concept real-time game included in this version of Ludii.

3
Modelling New Games

Game descriptions for Ludii are written in text files with a `.lud` extension. The language used to describe games for Ludii is defined by a *class grammar* (Browne, 2016) approach; it is automatically derived from the *ludeme* classes available in Ludii's `.jar` file. The full grammar is listed in Appendix C.

3.1 Modelling Games as Trees of Ludemes

The basic premise of the language is that ludemes are described as their name, followed by a whitespace-separated list of arguments, all wrapped up in a pair of parentheses:

Ludeme 1

```
(ludemeName arg1 arg2 arg3 ...)
```

Some of those arguments may themselves also be ludemes, leading to a tree of ludemes. The "outer" ludeme of a full game description is always the `game` ludeme. The following example shows the full game description of *Tic-Tac-Toe*:

Ludeme 2

```
(game "Tic-Tac-Toe"
  (mode 2)
  (equipment {
    (board (square 3) (square))
    (disc P1)
    (cross P2)
  }
  )
  (rules
```

```
      (play (to (mover) (empty)))
      (end (line length:3) (result Mover Win))
    )
  )
```

The three parameters for the `game` ludeme are `mode`, `equipment`, and `rules`.

3.1.1 Mode

The `mode` ludeme is used to define the basic control flow of a game. Alternating-move games are assumed to be the default type of games in Ludii. This means that, for alternating-move games, it is only necessary to supply the number of players as an argument to the `mode` ludeme. This is why the *Tic-Tac-Toe* description above usese `(mode 2)`; it is a two-player game.

For games that are not alternating-move games, an extra argument should be added to define the type of game. For example, *Rock-Paper-Scissors* uses `(mode 2 Simultaneous)`, and our proof-of-concept real-time *Tron* implementation uses `(mode 2 Realtime)`.

3.1.2 Equipment

The `equipment` ludeme is used to define equipment required to play a game. Broad categories of equipment are:

- **Components**: pieces (e.g. discs and crosses in *Tic-Tac-Toe*, pawns, kings, queens, etc. in *Chess*), cards, tiles, dice, etc.
- **Containers**: boards, hands, decks, pools, etc.
- **Other**: important regions (e.g. opposite sides of boards to be connected to each other for a victory in *Hex*), tracks to move along (in race games), etc.

For example, the *Tic-Tac-Toe* description above defines its board – a square board of size 3, tiled with square cells – and pieces to be placed by players – a disc for player 1, and a cross for player 2 – as equipment.

3.1.3 Rules

Finally, the `rules` ludeme is used to define the rules according to which a specific game is played. These are split up in two or three sets of rules; `start` (optional), `play` (required), and `end` (required).

- `start`: This can be used to define any rules to be applied whenever a new game starts. Defining `start` rules is optional; many games do not need any `start` rules. If a game has no `start` rules, any containers (such as boards, or player hands) will be completely empty when a game starts. Examples of games that require `start` rules are *Chess* or *Breakthrough*, because they require pieces to be set up on the board before gameplay can start.
- `play`: This is used to define how Ludii should generate its list of legal moves for any game state. Adding `play` rules is compulsory.
- `end`: This is used to define when Ludii should declare a game to be over, and what the outcome is (e.g. which player is a winner or loser, or how are the players ranked?). Adding `end` rules is compulsory.

For example, the *Tic-Tac-Toe* description above does not require any `start` rules – gameplay starts with an empty board. Every turn, the current mover in *Tic-Tac-Toe* chooses one empty cell of the board to place on of its pieces. This is defined by `(play (to (mover) (empty)))`. Because there is only one container (one board), and every player only owns a single type of piece, it is not required to explicitly define the type of piece to place – or which container to place them in – in the `play` rules. Whenever a player completes a line of three pieces, they win the game. This is defined by `(end (line length:3) (result Mover Win))`.

By default, Ludii declares any game in which no player is able to make a legal move a draw. It is not necessary to explicitly define this rule for *Tic-Tac-Toe* (or any other game).

3.2 Walkthrough: Implementing *Amazons* from Scratch

In this section, we provide a step-by-step guide for how a game like *Amazons* may be described for Ludii.

3.2.1 Step 1: A Minimum Legal Game Description

We start out with the minimum description that results in a legal game description that may be loaded in Ludii:

Ludeme 3

```
1  (game "Amazons"
2    (mode 2)
3    (equipment
4      (board (square 10) (square))
5    )
6    (rules
7      (play (byPiece))
8      (end (stalemated Next) (result Mover Win))
9    )
10 )
```

Line 2 defines that we wish to play a two-player game, where it is implied by default to be an alternating-move game. Lines 3–5 define that we wish to use a square board of size 10, tiled with squares. There are no `start` rules (yet). Line 7 uses one of the simplest `play` rules available in Ludii; `(byPiece)`, which simply defines that Ludii should loop through all pieces owned by a player, and extract legal moves from the piece types to generate the list of legal moves for a mover. Finally, line 8 defines that we expect the player who last made a move to win the game whenever the next player to move is *stalemated* (meaning that they cannot make any moves).

This game description can be successfully loaded into Ludii. However, it cannot be played yet. This is because we define legal moves to be extracted from pieces, but we did not yet define any piece types or add mechanisms to place pieces on the board!

3.2.2 Step 2: Defining Pieces and Placing Them on the Board

We extend the game description listed above by defining piece types, and adding `start` rules to place them on the board:

Ludeme 4

```
1  (game "Amazons"
2    (mode 2)
3    (equipment {
4      (board (square 10) (square))
5      (queen Each)
6      (dot None)
7    }
8    )
9    (rules
10     (start {
11       (place "Queen1" {"A4" "D1" "G1" "J4"})
12       (place "Queen2" {"A7" "D10" "G10" "J7"})
13     }
14     )
15     (play (byPiece))
16     (end (stalemated Next) (result Mover Win))
17   )
18 )
```

Line 5 defines that each player should have a piece of the `queen` type. Ludii will automatically label these as `"Queen1"` and `"Queen2"` for players 1 and 2, respectively. Additionally, in line 6 we define a `dot` piece, which is not owned by any player. This is the piece type that we will use in locations that players block by shooting their "arrows". Line 10–14 ensure that any game is started by placing objects of the two different types of queens in the correct starting locations. The labels used to specify these locations can be seen in Ludii by enabling "Show Coordinates" in Ludii's *View* menu.

This game description can be played in Ludii, but will immediately end in a win for player 1. This is because our `queen` piece types do not yet define any legal moves to be extracted by the `byPiece` ludeme. Therefore, the `end` condition immediately triggers at the end of the first turn – where player 1 is the `Mover`, and player 2 is the `Next` player, who is stalemated.

3.2.3 Step 3: Defining Queen Moves

In this step, we add a description of the moves to our definition of the `queen` types:

Ludeme 5

```
1  (game "Amazons"
2    (mode 2)
3    (equipment {
4      (board (square 10) (square))
5      (queen Each (slide (in (to) (empty))))
6      (dot None)
7    }
8    )
9    (rules
10     (start {
11       (place "Queen1" {"A4" "D1" "G1" "J4"})
12       (place "Queen2" {"A7" "D10" "G10" "J7"})
```

```
13      }
14      )
15      (play (byPiece))
16      (end (stalemated Next) (result Mover Win))
17    )
18  )
```

The only line that has been changed in comparison to the previous step is line 5. Inside the `queen` ludeme, we have added the following: `(slide (in (to) (empty)))`. This defines that the piece is permitted to slide along any axis of the used board, as long as we keep moving through locations that are empty. No additional restrictions – in terms of direction or distance, for example – are required for queen moves.

This game description is playable in Ludii, but gameplay cannot end up until a very high cap[4] on the number of turns has been reached. This is because, in this ruleset, no player will ever be stalemated.

3.2.4 Step 4: Adding the Final Rules for *Amazons*

To complete the game of *Amazons*, we need to allow players to shoot an arrow after moving a queen. This is implemented in the following game description:

Ludeme 6

```
1  (game "Amazons"
2    (mode 2)
3    (equipment {
4      (board (square 10) (square))
5      (queen Each (slide (in (to) (empty)) (then (replay))))
6      (dot None)
7    }
8    )
9    (rules
10     (start {
11       (place "Queen1" {"A4" "D1" "G1" "J4"})
12       (place "Queen2" {"A7" "D10" "G10" "J7"})
13     }
14     )
15     (play
16       (if (even (turn))
17         (byPiece)
18         (shoot (lastToMove) (in (to) (empty)) "Dot0")
19       )
20     )
21     (end (stalemated Next) (result Mover Win))
22   )
23 )
```

In line 5, we have appended `(then (replay))` in the `queen` moves. This means that, after any `queen` move, the same player gets to make another move. The `play` rules in lines 15–20

[4] Automatically determined by Ludii.

have been changed to no longer exclusively extract their moves from the pieces. Only in even turns (0, 2, 4, etc.) do we still make a `queen` move (using `(byPiece)`. In odd turns, the moves are defined by `(shoot (lastToMove) (in (to) (empty)) "Dot0")`. This rule lets us shoot a piece of type `"Dot0"` into any empty position, starting from the location that we last moved to – this is the location that our last `queen` move ended up in. This game description implements the full game of *Amazons* for Ludii.

3.2.5 Step 5: Using a Chessboard

The game description above plays correctly, but does not look appealing because it uses Ludii's default colours for the board. This can be easily improved by explicitly defining that we wish to use a chessboard (of size 10) in line 4:

Ludeme 7

```
1  (game "Amazons"
2    (mode 2)
3    (equipment {
4      (chessBoard 10)
5      (queen Each (slide (in (to) (empty)) (then (replay))))
6      (dot None)
7    }
8    )
9    (rules
10     (start {
11       (place "Queen1" {"A4" "D1" "G1" "J4"})
12       (place "Queen2" {"A7" "D10" "G10" "J7"})
13     }
14     )
15     (play
16       (if (even (turn))
17         (byPiece)
18         (shoot (lastToMove) (in (to) (empty)) "Dot0")
19       )
20     )
21     (end (stalemated Next) (result Mover Win))
22   )
23 )
```

3.3 Metadata

Following the definition of a `game` ludeme in a `.lud` file, additional pairs of strings may be provided as metadata. For instance, Ludii's *Amazons* file follows up the above listing with:

Ludeme 8

```
(metadata
  { "rules" "The game is played on a square grid, usually 10 by 10.
Each player starts with four amazons (which move as chess queens,
    but do not capture) placed in predetermined locations on their
    side of the board.
On a player turn, he moves one amazon and then fires an arrow from
    the arrival position.
Firing an arrow consists in placing a new arrow piece on the board;
     also the arrow moves like a queen from the position it is fired
    .
The landed arrow doesn't move anymore and creates a permanent block
     on the board. Arrows can not be passed through by either
    amazons or other arrows.
The first player who can not make a legal move (move and then shoot
    ) loses the game."}
)
```

Metadata can be used for a variety of purposes. For example, Ludii can detect metadata labelled with `"rules"` as in the listing above, and print the text to the Info tab of the user interface. Metadata may also be used to adjust the default colours used by the user interface for a game. For example, the *Hex* implementation in Ludii contains the following metadata:

Ludeme 9

```
    { "P1Colour" "Red" }
    { "P2Colour" "Blue" }
    { "Fill0Colour" "Very Light Grey" }
    { "LineColour" "Light Grey" }
```

3.4 Options

For many games, there exists a wide selection of variants that would have nearly identical game descriptions if they were all implemented as separate `.lud` files. To avoid writing many such files, we allow for a single game description to contain many variants of a game through a mechanism referred to as *options*. Options are defined outside of the `(game ...)` ludeme in a `.lud` file, but typically used inside it. The basic syntax for the definition of a single value for an option is given by:

Ludeme 10

```
(option <ID> "Name" <Value>)
```

The `ID` is simply expected to be an identifier for the option; 1 for the first option, 2 for the second, etc. `Name` is the name of a particular value for an option, and used in the menu of the user interface that can be used by users to change the variant of the game to be played. `Value` is the value that will be plugged into the game description used when this option value is selected.

For example, the *Hex* game description file included in Ludii contains the following definitions for options:

Ludeme 11

```
(option <1> "Board Size/3x3" <3>)
(option <1> "Board Size/4x4" <4>)
(option <1> "Board Size/5x5" <5>)
(option <1> "Board Size/6x6" <6>)
(option <1> "Board Size/7x7" <7>)
(option <1> "Board Size/8x8" <8>)
(option <1> "Board Size/9x9" <9>)
(option <1> "Board Size/10x10" <10>)*
(option <1> "Board Size/11x11" <11>)**
(option <1> "Board Size/12x12" <12>)
(option <1> "Board Size/13x13" <13>)
(option <1> "Board Size/14x14" <14>)*
(option <1> "Board Size/15x15" <15>)
(option <1> "Board Size/16x16" <16>)
(option <1> "Board Size/17x17" <17>)*
(option <1> "Board Size/18x18" <18>)
(option <1> "Board Size/19x19" <19>)

(option <2> "End/Normal" <Win>)*
(option <2> "End/Misere" <Loss>)
```

The first seventeen lines all define different possible values for the "Board Size" option, which has been given an identifier of 1. We support all integer values ranging from 3 to 19 for this option. In the user interface, these values are named 3×3, 4×4, etc., and can all be found under a submenu named "Board Size". Ludii uses the value 11 as a default value for this option, because this value is followed by the greatest number of asterisks in the definition.

The last two lines define values for a second option, which may be used to modify the win condition of the game. Note that these values are for a new option, with a new identifier of 2. The two possible values ("Normal" and "Misere") can be found in a different submenu, named "End", in Ludii's user interface.

The full game description for *Hex* is given by the listing below. Note that all the options, as shown in the listing above, should be appended after this listing; in the same `.lud` file, but *after* the `(game ...)` ludeme.

Ludeme 12

```
1  (game "Hex"
2    (mode 2)
3    (equipment {
4      (HexBoard <1>)
5      (ball Each)
6      (regions P1 { (edge NE) (edge SW) } )
7      (regions P2 { (edge NW) (edge SE) } )
8    }
9    )
10   (rules
```

```
11      (play (to (empty)))
12      (end (connect Mover) (result Mover <2>))
13    )
14  )
```

In this listing, line 4 contains the `<1>` token, and line 12 contains the `<2>` token. When Ludii loads the game, these tokens are automatically replaced by the selected option values for options 1 and 2, respectively. Recall that the default values are 11 for the board size, and `Win` for the end rule. This means that, by default, Ludii will interpret line 4 as `(HexBoard 11)`, and line 12 as `(end (connect Mover) (result Mover Win))`.

3.5 Defines

While the options mechanism above helps to avoid (near-)duplicate descriptions for variants of games, it does not help to reduce duplication of partial descriptions within a single game description. For this purpose, we users to *define* "macros", such that a frequently-repeated part of a game description can be reproduced multiple times using a single label.

For example, the *Breakthrough* game description included in Ludii contains the following *define*:

Ludeme 13

```
(define "Moves" (or {
                    (step F (in (to) (empty)))
                    (step ForwardDiagonal (not (isFriend (who (to))
                        )) (remove (to)))
                })
)
```

Such a *define* can be placed anywhere inside a game description. In this define, `"Moves"` is the label, and everything following it is the part of the game description that we aim to reproduce multiple times. In this specific case, it is ludeme that defines movement rules for *Breakthrough*: pieces are allowed to stop forwards into empty locations, or diagonally forwards into locations that are not occupied by friendly pieces. The latter type of move has the additional consequence that it removes whatever piece may previously have been in that location. Any occurrence of the string `"Moves"` that follows this *define* in the game description will automatically be replaced by the complete `(or ...)` ludeme in the *define*.

This *define* allows the piece types in *Breakthrough* to be subsequently defined in a succinct manner:

Ludeme 14

```
(pawn P1 N "Moves")
(pawn P2 S "Moves")
```

Without using a *define*, the full ludeme defining legal moves would have to be reproduced for each of these pieces.

Defines in Ludii game descriptions may also be parameterised. For example, the end rules of *Breakthrough* are defined using a parameterised *define*:

Ludeme 15

```
(define "End" (end (in (lastToMove) #1) (result #2 Win)))
(or { ("End" (top) P1) ("End" (bottom) P2) } )
```

In the *define* in the first line of this listing, `#1` and `#2` denote two parameters. Whenever this *define* is "invoked" (i.e. whenever its label, `"End"`, is used in a game description), it must be followed up by arguments – in this case two – to fill in the parameters. Note that, when a *define* with parameters is invoked, it is wrapped up in an extra set of parentheses (together with its parameters). The above listing could equivalently be written without *defines* as follows:

Ludeme 16

```
(or {
  (end (in (lastToMove) (top)) (result P1 Win))
  (end (in (lastToMove) (bottom)) (result P2 Win))
})
```

3.6 Creating Your Own Games

While Appendix C fully specifies the syntax to use for correct game descriptions, it does not define the semantics of any game description. This requires a full documentation of every ludeme included in Ludii, and all of their parameters. We aim to have this ready for the first full release of Ludii, but this is not yet ready for the pre-release.

In the meantime, we recommend extracting the `.lud` files bundled with the Ludii download from its `.jar` archive, and using them as examples. Every single game currently playable in Ludii has such a `.lud` file. We hope that all of these examples will be sufficient to get you started creating similar games. Many of the ludemes have easily-understandable English names, and directly correspond to simple, atomic game concepts. This should help to lower the barrier of entry.

4 Implementing Custom Agents

Ludii's `.jar` file provides an abstract `AI` class which may be extended by third-party users to develop their own AI players. We first provide instructions for how to develop your own AI for Ludii, and describe how to load them into the Ludii application afterwards. Finally, we explain how to run Ludii games programmatically (circumventing its graphical user interface), which we expect to be particularly interesting for AI developers and AI researchers. Many of these instructions can also be found on https://github.com/Ludeme/LudiiExampleAI, which hosts a repository with some example AIs for Ludii.

4.1 Implementing a Ludii AI Player

Ludii's `.jar` file cannot only be used to run the application, but also as a library for other `Java` projects. Any `Java` project that uses it as a library can implement new AI players for Ludii by creating a class that extends Ludii's `util.AI` abstract class.

The most important abstract method to be overridden from this class is `selectAction`, for which the signature is given in the following box:

Java 1

```
public abstract Move selectAction
(
    final Game game,
    final Context context,
    final double maxSeconds,
    final int maxIterations,
    final int maxDepth
);
```

Any AI for Ludii is expected to implement this method such that it returns the move to be played in a given game state. The `game` object is an object that contains the rules of the loaded

game, and methods required for implementing a forward model (such as methods to obtain the list of legal moves in a game state, or methods to apply a move or roll out a full playout). The `context` object contains data for the current game state, the full list of moves applied previously, etc. The final three parameters may be used to restrict an AI player's "thinking time" (in seconds), maximum number of iterations (for instance, MCTS iterations), and search depth. Ludii does not currently enforce any such restrictions, but we expect they may be useful for AI researchers to run experiments. All of Ludii's built-in AIs respect the maximum number of seconds, and all MCTS-based built-in AIs also respect restrictions on their iteration count. Of course, future competitions run using Ludii (Stephenson, Piette, Soemers, & Browne, 2019) will more strictly enforce such restrictions.

An example implementation, taken from the example "Random AI" from our example repository, is given in the following box:

Java 2

```
1  public Move selectAction
2  (
3      final Game game,
4      final Context context,
5      final double maxSeconds,
6      final int maxIterations,
7      final int maxDepth
8  )
9  {
10     FastArrayList<Move> legalMoves = game.moves(context).moves();
11
12     if (legalMoves.isEmpty())
13     {
14         final Move passMove = new Move(new ActionPass());
15         passMove.setMover(player);
16         return passMove;
17     }
18
19     if
20     (
21         context.model() instanceof SimultaneousMove ||
22         context.model() instanceof RealTime
23     )
24     {
25         legalMoves = AIUtils.extractMovesForMover(legalMoves, player);
26     }
27
28     final int r =
29             ThreadLocalRandom.current().nextInt(legalMoves.size());
30     return legalMoves.get(r);
31 }
```

In lines 12-17, we handle the special case where no legal moves are available by simply passing. In lines 19-26, we extract only those legal moves that belong to the player given by the `player` variable. This is required in simultaneous-move games, and real-time games, because those games may simultaneously have different legal moves for different players at any given point in

time.

The `player` variable mentioned above is how the AI knows which player it is currently controlling. This value is supplied to AIs in an `initAI()` method which is called whenever an AI starts playing a new game, and can be overridden to perform any desired initialisation. For example, the example Random AI implements it as follows:

Java 3

```
public void initAI(final Game game, final int playerID)
{
    this.player = playerID;
}
```

Ludii's `AI` abstract class also has a `String` member named "`friendlyName`", which may sometimes be displayed in the GUI. The example Random AI sets this property in its constructor:

Java 4

```
public RandomAI()
{
    this.friendlyName = "Example Random AI";
}
```

Finally, the class has a method that may be overridden to inform the Ludii application if a certain (type of) game cannot be played by an AI. For example, our example implementation of a simple UCT agent[5] (Browne et al., 2012) overrides this method as follows:

Java 5

```
public boolean supportsGame(final Game game)
{
    final int stateFlags = game.stateFlags();

    if ((stateFlags & StateType.Stochastic) != 0)
        return false;

    if ((stateFlags & StateType.Simultaneous) != 0)
        return false;

    return true;
}
```

This implementation ensures that, if the agent is loaded into the Ludii application, it will never try to make them play any games with stochasticity or simultaneous moves; instead, Ludii will automatically replace it with one of its own built-in AIs that does support those types of games.

[5] https://github.com/Ludeme/LudiiExampleAI/blob/master/src/mcts/ExampleUCT.java

4.2 Loading Third-party AI Players in Ludii

Ludii can import any custom AI controller that extends Ludii's `AI` abstract class from `.jar` files. The steps to do this are:

1. Create a `.jar` file containing the `.class` files for your AI implementation, or obtain one from a third party. **Note**: as with third-party `.lud` game description files, we recommend only using files from trusted sources!
2. Select "From JAR" for one or more players in the Player Preferences dialog.
3. Navigate to, and select, your `.jar` file in the file chooser that pops up.
4. A dialog will pop up with a drop-down menu listing all the AI classes that were found in the selected `.jar` file. Whichever class is selected from this menu will be used as an AI controller by Ludii.

4.3 Programmatically Running Ludii Games

Apart from loading custom AI controllers into the Ludii application, it is also possible to programmatically run Ludii games from any Java project using Ludii's `.jar` file as a library. We expect this to be particularly convenient for AI developers to test their implementations during development without having to use the GUI, and for AI researchers to run large-scale experiments.

Ludii's repository with example AIs also contains a Tutorial file with extensive comments, which showcases how some of the basic functionality of Ludii may be accessed programmatically. This file can be found at the using the following URL: https://github.com/Ludeme/LudiiExampleAI/blob/master/src/experiments/Tutorial.java. Two additional, shorter examples for running games programmatically can be found at:

- https://github.com/Ludeme/LudiiExampleAI/blob/master/src/experiments/RunLudiiMatch.java
- https://github.com/Ludeme/LudiiExampleAI/blob/master/src/experiments/RunCustomMatch.java

5 Troubleshooting

5.1 Cannot Get Ludii to Run

If double-clicking Ludii's `.jar` file does not make the application run, it is possible that your operating system is not using the correct executable to run it. See https://stackoverflow.com/questions/8511063/how-to-run-jar-file-by-double-click-on-windows-7-64-bit for Windows, or search for similar solutions for other operating systems.

Alternatively, if you just installed (a new version of) Java, it may help to unplug and replug your monitor.

5.2 Poor Performance User Interface

If Ludii appears slow or unresponsive on your device then we recommend the following.

- Ensure you have Java 8 or higher installed (preferably the latest version).
- Turn off high-DPI scaling on your device. Depending on your operating system, this can be achieved as follows:
    1. Windows: Open "Display settings" by searching for it in the Windows search bar, or by right clicking on the desktop. Under the "Scale and layout" section, change the option labelled "Change the size of text, apps, and other items" to 100%.
    2. Windows (advanced): This option is more complicated, but will ensure that high-DPI scaling is only removed from Java applications. Navigate to you Java installation directory, and locate the file "javaw.exe" inside the "bin" directory. Right click on this file, select "Properties", select "Compatibility", select "Change high DPI settings" if present, activate High DPI scaling override and override the scaling behaviour performed by System.
    3. macOS: Open "System preferences" and select "Displays". Under "Resolution" select "Default for display".

4. Ubuntu: Open “Settings” and select “Screen Display”. Change the Scale option to 100%.

Acknowledgement

This work is part of the *Digital Ludeme Project*, funded by €2m European Research Council (ERC) Consolidator Grant #771292, being run by Cameron Browne at Maastricht University's Department of Data Science and Knowledge Engineering over 2018–23.

A
Keyboard Shortcuts

Keystroke(s)	Action
Ctrl+L	Load Game
Ctrl+F	Load Game from File
Ctrl+S	Save Trial
Ctrl+G	Generate Grammar
Ctrl+R	Restart Game
Ctrl+M	Random Move
Ctrl+P	Random Playout
Ctrl+T	Time Random Playouts
Alt+S	Game Screenshot
Ctrl+B	Show Board
Ctrl+D	Show Graph
Ctrl+A	Show Axes
Alt+M	Show Possible Moves
Alt+L	Show Last Move
Alt+I	Show Cell Indices
Alt+E	Show Edge Indices
Alt+F	Show Face Indices
Alt+C	Show Coordinates
Alt+A	Show AI Distribution
Alt+C	Colour Preferences

Alt+P	Player Preferences
Alt+Z	Puzzle Preferences
Ctrl+0	Show track shared by all players
Ctrl+1	Show track for Player 1
Ctrl+2	Show track for Player 2
...	...
Ctrl+8	Show track for Player 8
Alt+N	Connect
Space	Start playing / Pause
Down arrow	Jump back to start
Left arrow	Go back one step
Right arrow	Go forwards one step
Up arrow	Jump forwards to end of game (so far)

B
Technical Details Built-in AIs

This appendix provides technical details for the different built-in AI controllers of Ludii.

B.1 Monte Carlo (flat)

This is a flat Monte-Carlo search agent. Let $\mathcal{M}(s)$ denote the set of legal actions for the agent in a game state s. For as many iterations as possible, it runs full playouts from the current game state s, where all moves are selected uniformly at random. At the end of every playout, the outcome is mapped to a value in $[-1, 1]$. Whichever move $m \in \mathcal{M}(s)$ resulted in the greatest average score among all playouts in which it was selected at the start of the playout is finally selected as move to play in s.

B.2 UCT

This is a fairly standard implementation of the UCT variant of MCTS (Browne et al., 2012). Some implementation details:

- The standard UCB1 policy (Auer, Cesa-Bianchi, & Fischer, 2002) is used in the MCTS selection phase, with an exploration constant $C = \sqrt{2}$ (i.e. equivalent to the standard formulation of UCB1 without an exploration hyperparameter).
- Moves in playouts are selected uniformly at random.
- The search tree is expanded by one node per iteration.
- Relevant parts of the search tree from a previous search process are reused when a new search process starts.
- In deterministic games, game states are stored in the nodes. For nondeterministic games, we use an "open-loop" approach (Perez, Dieskau, Hünermund, Mostaghim, & Lucas, 2015); game states are not stored in nodes, but re-generated (by a possibly nondeterministic forward model) as we traverse from a root node to a leaf node in every iteration.

- The selection phase may select children that have already been visited before, even if there are also children without any visits available. The "mean value" estimate for children without any visits is equal to the value estimate of their parent node.
- The final move selected by the algorithm is that corresponding to the child of the root node with the highest visit count (the "robust child").
- The implementation maps all game outcomes to lie in $[-1, 1]$ for its value estimates.
- No transposition table is used.
- This implementation does not support simultaneous-move games.

B.3 MC-GRAVE

An implementation of Generalized Rapid Action Value Estimation (GRAVE) (Cazenave, 2015). Most implementation details are identical to those of UCT, as described above. Differences are:

- The tree is traversed using GRAVE's strategy in the selection phase of MCTS. We use $ref = 100$ (this is the threshold determining which node's AMAF values are used), and $bias = 10^{-6}$ (this is the hyperparameter used in the denominator when computing the β_m weight for GRAVE). These hyperparameter values are loosely based on the experiments reported by Cazenave (2015).
- The selection phase does not incorporate any explicit exploration terms. In the future we may add a UCT-GRAVE variant with an explicit exploration term (this is why we named the current implementation MC-GRAVE instead of GRAVE).
- The value estimate for unvisited children is set to $10,000$ (deliberately far outside the normal range of value estimates, which is upper bounded by 1.0), instead of setting it to be equal to the value estimate of the parent node. This is required due to the lack of an explicit exploration term in MC-GRAVE.
- AMAF statistics are updated once *per occurrence* of a move in a playout. This means that, if the same move (or one with the same hash code) occurred more than once in a single playout, the AMAF statistics for that move are updated more than once for a single playout.

B.4 Biased MCTS

A variant of MCTS that is biased by *features* (local patterns for state-action pairs). Most implementation details are identical to those of UCT, as described above. Differences are:

- The tree is traversed using the same PUCT strategy as AlphaGo Zero (Silver et al., 2017) in the selection phase of MCTS. We use an exploration constant of 2.5.
- The policy used to bias PUCT is a softmax function of logits – one logit per state action pair, which is computed as the dot product between a vector of weights and a vector of binary features for the state-action pair (Browne, Soemers, & Piette, 2019; Soemers, Piette, & Browne, 2019).
- The same policy is also used to select moves in the playout phase of MCTS.

In the pre-release version of Ludii, Biased MCTS only supports a small selection of games. This is because, in this version, we only include a small selection of handcrafted features and weights.

Later versions of Ludii will include automatically learned features and weights, ideally for all games. Before we can do this, we are still working on scaling up some of our previous work on learning these features (Soemers et al., 2019) to larger collections of (types of) games. We also aim to work on automatically identifying and removing unimportant features after such a learning process, since our previous work only focussed on continuously growing the feature set – which in some games reduces performance due to increased computational overhead. Games support by Biased MCTS in this pre-release version of Ludii are:

- *Ard Ri*.
- *Chess*.
- *Double Chess (Skirmish)*.
- *Gomoku*.
- *Half Chess*.
- *Hex*.
- *Hnefatafl Fetlar*.
- *Reversi*.
- *Skirmish*.
- *Yavalath*.

C
Ludii's Class Grammar

The full class grammar of Ludii's pre-release version, which defines the language in which games can be described, is given in the following listing. Note that the grammar is automatically derived from the library of ludemes in the Ludii application, and may therefore change whenever Ludii is updated.

Grammar 1

```
//-----------------------------------------------------------------------------
// game

<game>       ::= (game <string> <mode> <equipment> <rules>)

//-----------------------------------------------------------------------------
// game.mode

<mode>       ::= (mode <player> [<modeType>] [<playout>]) |
                 (mode <integer> [<modeType>] [<playout>]) |
                 (mode {<player>} [<modeType>] [<playout>])
<modeType>   ::= Alternating | Simultaneous | Realtime

//-----------------------------------------------------------------------------
// game.mode.player

<player>     ::= (player [<string>] [<string>] [<compassDirection>] [{<integer
    >}])

//-----------------------------------------------------------------------------
// game.equipment.container.tiling.directions

<compassDirection> ::= N | NE | E | SE | S | SW | W | NW
```

```
<directionChoice> ::= None | All | Orthogonal | Diagonal | Axial | Column |
                      Row | N | E | S | W | NE | SE | NW | SW | Ahead | F |
                      ForwardDiagonal | OrthogonalNotBackward |
                      OrthogonalAndForwardDiagonal | ForwardAndDiagonal | Right |
                      Left | SameDirection | SameDirectionDiagonal |
                      OppositeDirection | CW | CCW | IN | OUT

//------------------------------------------------------------------------------
// game.playout

<playout>    ::= <model> | <addToEmpty> | <byPiecePlayout>
<addToEmpty> ::= (addToEmpty)
<byPiecePlayout> ::= (byPiecePlayout)

//------------------------------------------------------------------------------
// game.mode.model

<model>      ::= <alternatingMove> | <realTime> | <simultaneousMove>
<alternatingMove> ::= (alternatingMove)
<realTime>   ::= (realTime)
<simultaneousMove> ::= (simultaneousMove)

//------------------------------------------------------------------------------
// game.equipment

<equipment>  ::= (equipment <item>) | (equipment {<item>})
<item>       ::= <component> | <container> | <map> | <regions> | <track>

//------------------------------------------------------------------------------
// game.equipment.component

<component>  ::= <card> | <deck> | <die> | <letter> | <number> | <dominoes> |
                 <piece>
<die>        ::= (die [<string>] <roleType> <integer> [<boolean>] [<integer>] [<
    compassDirection>] [<moves>])
<letter>     ::= (letter [<string>] <roleType> [<string>] [<integer>] [<
    compassDirection>] [<moves>])
<number>     ::= (number [<string>] <roleType> <integer> [<integer>] [<
    compassDirection>] [<moves>])

//------------------------------------------------------------------------------
// game.equipment.component.card

<card>       ::= (card [<string>] <roleType> <cardType> <integer>)
<deck>       ::= (deck [<string>] [<roleType>] <integer>)

//------------------------------------------------------------------------------
// game.rules.play.moves

<moves>      ::= <and> | <byPiece> | <byPieceType> | <captureByApproach> |
```

```
                <captureByWithdraw> | <checkmate> | <checkMove> | <fromTo> |
                <hop> | <leap> | <pass> | <promotion> | <select> | <shoot> |
                <slide> | <step> | <to> | <constraints> | <flanked> |
                <flip> | <forDirn> | <if> | <observe> | <or> | <pending> |
                <priority> | <remove> | <replay> | <roll> | <setCounter> |
                <setDouble> | <setOwner> | <setScore> | <setState> | <sow> |
                <surrounded>
<and>         ::= (and <moves> <moves> [<then>]) | (and {<moves>} [<then>])
<byPiece>     ::= (byPiece [<string>] [<moves>] [<int>] [container:<int>] [top:<
    boolean>] [<then>])
<byPieceType> ::= (byPieceType [<then>])
<captureByApproach> ::= (captureByApproach [<int>] [<directionChoice>] [<boolean
    >] [<moves>] [<then>])
<captureByWithdraw> ::= (captureByWithdraw [<int>] [<directionChoice>] [<boolean
    >] [<moves>] [<then>])
<checkmate>   ::= (checkmate <int> [<then>])
<checkMove>   ::= (checkMove <moves> <boolean> [<then>])
<flanked>     ::= (flanked [<int>] [<directionChoice>] [<int>] [<boolean>] [<
    boolean>] [<moves>] [<then>])
<flip>        ::= (flip [<int>] [<then>])
<forDirn>     ::= (forDirn [<int>] [<directionChoice>] [<int>] <boolean> <moves>
    [<then>])
<if>          ::= (if <boolean> <moves> [<moves>] [<then>])
<observe>     ::= (observe <int> [<int>] [<then>])
<or>          ::= (or <moves> <moves> [<then>]) | (or {<moves>} [<then>])
<pending>     ::= (pending [<then>])
<priority>    ::= (priority {<moves>} [<then>]) |
                (priority <moves> <moves> [<then>])
<remove>      ::= (remove <int> [<then>])
<replay>      ::= (replay)
<roll>        ::= (roll [<int>] [<then>])
<setCounter>  ::= (setCounter [<int>] [<then>])
<setDouble>   ::= (setDouble [<int>] [<then>])
<setOwner>    ::= (setOwner [<variableType>] <region> <int> [<then>])
<setScore>    ::= (setScore {<roleType>} {<int>} [<then>]) |
                (setScore {<int>} {<int>} [<then>]) |
                (setScore <roleType> <int> [<then>]) |
                (setScore <int> <int> [<then>])
<setState>    ::= (setState <int> <int> [<then>])
<sow>         ::= (sow <int> <int> <string> [<boolean>] [<moves>] [included:<
    boolean>] [exception:<int>] [recursive:<boolean>] [<then>])
<surrounded>  ::= (surrounded [<int>] [<directionChoice>] [<boolean>] [<boolean>]
    [<moves>] [<then>])
<then>        ::= (then <moves>)

//------------------------------------------------------------------------------
// game.functions.region

<region>      ::= (region <int> [<string>])
<allSites>    ::= (allSites [<int>] [<boolean>])
```

```
<around>     ::= (around <region> [<regionType>] [<int>] [<directionChoice>] [<
    boolean>] [included:<boolean>] [<variableType>])
                 |
                 (around <int> [<regionType>] [<int>] [<directionChoice>] [<
                     boolean>] [included:<boolean>] [<variableType>])
<borderRegion> ::= (borderRegion <int>)
<bottom>     ::= (bottom [<int>])
<centre>     ::= (centre [<int>])
<column>     ::= (column <int>)
<corners>    ::= (corners [<int>])
<difference> ::= (difference <region> <region>)
<directionRegion> ::= (directionRegion <int> [<directionChoice>] [<boolean>])
<edge>       ::= (edge [<int>] <direction>)
<edgeFace>   ::= (edgeFace <int>)
<edgeVertex> ::= (edgeVertex <int>)
<empty>      ::= (empty [<int>] [<variableType>])
<exteriors>  ::= (exteriors [<int>])
<facesEdge>  ::= (facesEdge <int> <int>)
<hand>       ::= (hand <int>)
<hintRegion> ::= (hintRegion)
<if>         ::= (if <region> <boolean>)
<occupied>   ::= (occupied [<int>])
<own>        ::= (own <int> [<string>] [<int>] [<string>] [top:<boolean>] [<
    variableType>])
<phase>      ::= (phase <region> <integer> <integer>)
<playable>   ::= (playable [<int>])
<row>        ::= (row <integer>)
<set>        ::= (set {<integer>}) | (set <integer>)
<stateRegion> ::= (stateRegion <int> [<string>] [<int>] [<string>] [<variableType
    >])
<top>        ::= (top [<int>])
<union>      ::= (union <region> <region>) |
                 (union {<region>})
<verticeFace> ::= (verticeFace <int>)
<walk>       ::= (walk [<int>] [<boolean>] {{<stepType>}})

//-------------------------------------------------------------------------------
// game.functions.other

<hint>       ::= (hint {<integer>} [<integer>]) | (hint <integer> [<integer>])
<values>     ::= (values <integer> <integer>)

//-------------------------------------------------------------------------------
// game.functions.booleans.puzzleConstraints

<allDifferent> ::= (allDifferent [<region>] [excepts:{<int>}]) | (allDifferent [<
    region>] [except:<int>])
<connected>  ::= (connected <areaType>)
<count>      ::= (count <region> [<int>] <int> [<variableType>])
<crossing>   ::= (crossing [<int>])
```

```
<equalParities> ::= (equalParities [<region>]) |
                    (equalParities [<string>])
<forEach>    ::= (forEach <variableType> <boolean>)
<mult>       ::= (mult [<region>] <int>) |
                    (mult <string> <int>)
<shape>      ::= (shape [<region>] <shapeType> [<variableType>]) |
                    (shape [<string>] <shapeType> [<variableType>])
<solved>     ::= (solved)
<sum>        ::= (sum [<region>] <int>) |
                    (sum <string> <int>)
<unique>     ::= (unique)

//-------------------------------------------------------------------------
// game.rules.play.moves.decision

<fromTo>     ::= (fromTo <region> <int> [levelTo:<int>] [rule:<boolean>] [<
    boolean>] [<moves>] [<roleType>] [stack:<boolean>] [<then>])
                    |
                    (fromTo <int> [levelFrom:<int>] <region> [count:<int>] [rule:<
                        boolean>] [<boolean>] [<moves>] [<roleType>] [stack:<boolean
                        >] [<then>])
                    |
                    (fromTo <region> <region> [rule:<boolean>] [<boolean>] [<moves>]
                         [<roleType>] [stack:<boolean>] [<then>])
                    |
                    (fromTo <int> [levelFrom:<int>] <int> [levelTo:<int>] [count:<
                        int>] [rule:<boolean>] [<boolean>] [<moves>] [<roleType>] [
                        stack:<boolean>] [<then>])
<hop>        ::= (hop [<int>] [<directionChoice>] [lead:<int>] [minRun:<int>] [
    maxRun:<int>] [after:<int>] <boolean> <boolean> [<moves>] [stopRule:<boolean
    >] [captureEffect:<moves>] [next:<moves>] [stack:<boolean>] [<then>])
<leap>       ::= (leap [<int>] <region> <boolean> [<moves>] [<moves>] [<then>])
<pass>       ::= (pass [<then>])
<promotion>  ::= (promotion [<int>] <string> [<then>]) |
                    (promotion [<int>] {<string>} [<then>])
<select>     ::= (select <region> [<then>]) |
                    (select <int> [<then>])
<shoot>      ::= (shoot [<int>] [<directionChoice>] [<int>] <boolean> <string> [<
    moves>] [<then>])
<slide>      ::= (slide [<int>] [<string>] [<directionChoice>] [min:<int>] [limit
    :<int>] <boolean> [<boolean>] [<moves>] [next:<moves>] [<then>])
<step>       ::= (step [<int>] [<directionChoice>] <boolean> [<moves>] [next:<
    moves>] [stack:<boolean>] [<then>])
                    |
                    (step <region> [<directionChoice>] <boolean> [<moves>] [next:<
                        moves>] [stack:<boolean>] [<then>])
<to>         ::= (to [<int>] [{<int>}] [state:<int>] <region> [<boolean>] [
    onStack:<boolean>] [<variableType>] [<then>])
                    |
```

```
                (to [<int>] [{<int>}] [state:<int>] <int> [<boolean>] [onStack:<
                    boolean>] [<variableType>] [<then>])

//-------------------------------------------------------------------------
// game.rules.play.moves.decisionPuzzle

<constraints> ::= (constraints {<boolean>} [<then>]) |
                (constraints <boolean> [<then>])

//-------------------------------------------------------------------------
// game.equipment.component.piece.large

<domino>     ::= (domino [<string>] <roleType> <integer> <integer>)
<dominoes>   ::= (dominoes <integer> <integer>)
<largePiece> ::= (largePiece [<string>] <roleType> {{<stepType>}} [<moves>]) |
                <domino> | <lPiece>
<lPiece>     ::= (lPiece [<string>] <roleType> [<moves>])

//-------------------------------------------------------------------------
// game.equipment.component.piece

<piece>      ::= <animal> | <arrow> | <ball> | <checkers> | <chess> | <cross> |
                <disc> | <dominoHalf> | <dot> | <hands> | <hex> |
                <largePiece> | <pill> | <senetPiece> | <shogi> | <square> |
                <pieceState> | <stratego> | <tafl> | <triangle> | <urPiece> |
                <xiangqi>
<arrow>      ::= (arrow [<string>] <roleType>)
<ball>       ::= (ball [<string>] <roleType> [<integer>] [<compassDirection>] [
    value:<integer>] [<moves>])
<cross>      ::= (cross [<string>] <roleType> [<integer>])
<disc>       ::= (disc [<string>] <roleType> [<integer>] [<compassDirection>] [
    value:<integer>] [scale:<string>] [<moves>])
<dominoHalf> ::= (dominoHalf [<string>] <roleType> [value:<integer>])
<dot>        ::= (dot [<string>] <roleType> [<integer>] [<moves>])
<hex>        ::= (hex [<string>] <roleType> [north:<boolean>])
<pill>       ::= (pill [<string>] <roleType>)
<senetPiece> ::= (senetPiece [<string>] <roleType> [<moves>])
<square>     ::= (square [<string>] <roleType> [<integer>])
<triangle>   ::= (triangle [<string>] <roleType>)
<urPiece>    ::= (urPiece [<string>] <roleType> [<moves>])

//-------------------------------------------------------------------------
// game.equipment.component.piece.animal

<animal>     ::= <camel> | <cat> | <cow> | <dog> | <elephant> | <goat> |
                <horse> | <leopard> | <rabbit> | <sheep> | <tiger> | <wolf>
<camel>      ::= (camel [<string>] <roleType> [value:<integer>] [<moves>])
<cat>        ::= (cat [<string>] <roleType> [value:<integer>] [<moves>])
<cow>        ::= (cow [<string>] <roleType> [<moves>])
<dog>        ::= (dog [<string>] <roleType> [value:<integer>] [<moves>])
```

```
<elephant>   ::= (elephant [<string>] <roleType> [value:<integer>] [<moves>])
<goat>       ::= (goat [<string>] <roleType> [<moves>])
<horse>      ::= (horse [<string>] <roleType> [value:<integer>] [<moves>])
<leopard>    ::= (leopard [<string>] <roleType> [<moves>])
<rabbit>     ::= (rabbit [<string>] <roleType> [<compassDirection>] [value:<
    integer>] [<moves>])
<sheep>      ::= (sheep [<string>] <roleType> [<compassDirection>] [<moves>])
<tiger>      ::= (tiger [<string>] <roleType> [<moves>])
<wolf>       ::= (wolf [<string>] <roleType> [<moves>])

//-------------------------------------------------------------------------
// game.equipment.component.piece.checkers

<checkers>   ::= <dame> | <man> | <manStar>
<dame>       ::= (dame [<string>] <roleType> [<compassDirection>] [<moves>])
<man>        ::= (man [<string>] <roleType> [<integer>] [<compassDirection>] [
    value:<integer>] [scale:<string>] [<moves>])
<manStar>    ::= (manStar [<string>] <roleType> [<compassDirection>] [<moves>])

//-------------------------------------------------------------------------
// game.equipment.component.piece.chess

<chess>      ::= <bishop> | <king> | <knight> | <mann> | <pawn> | <queen> |
                 <rook>
<bishop>     ::= (bishop [<string>] <roleType> [<moves>])
<king>       ::= (king [<string>] <roleType> [<moves>])
<knight>     ::= (knight [<string>] <roleType> [<compassDirection>] [<moves>])
<mann>       ::= (mann [<string>] <roleType> [<moves>])
<pawn>       ::= (pawn [<string>] <roleType> [<compassDirection>] [<moves>])
<queen>      ::= (queen [<string>] <roleType> [<moves>])
<rook>       ::= (rook [<string>] <roleType> [<moves>])

//-------------------------------------------------------------------------
// game.equipment.component.piece.hands

<hands>      ::= <hand> | <paper> | <rock> | <scissors>
<hand>       ::= (hand <string> <roleType> right:<boolean> <integer> [<moves>])
<paper>      ::= (paper <roleType>)
<rock>       ::= (rock <roleType>)
<scissors>   ::= (scissors <roleType>)

//-------------------------------------------------------------------------
// game.equipment.component.piece.shogi

<shogi>      ::= <fuhyo> | <ginsho> | <hisha> | <kakugyo> | <keima> |
                 <kinsho> | <kyosha> | <narigin> | <narikei> | <narikyo> |
                 <osho> | <ryuma> | <ryuo> | <tokin>
<fuhyo>      ::= (fuhyo <roleType> [<compassDirection>] [<moves>])
<ginsho>     ::= (ginsho <roleType> [<compassDirection>] [<moves>])
<hisha>      ::= (hisha <roleType> [<compassDirection>] [<moves>])
```

```
<kakugyo>    ::= (kakugyo <roleType> [<compassDirection>] [<moves>])
<keima>      ::= (keima <roleType> [<compassDirection>] [<moves>])
<kinsho>     ::= (kinsho <roleType> [<compassDirection>] [<moves>])
<kyosha>     ::= (kyosha <roleType> [<compassDirection>] [<moves>])
<narigin>    ::= (narigin <roleType> [<compassDirection>] [<moves>])
<narikei>    ::= (narikei <roleType> [<compassDirection>] [<moves>])
<narikyo>    ::= (narikyo <roleType> [<compassDirection>] [<moves>])
<osho>       ::= (osho <roleType> [<compassDirection>] [<moves>])
<ryuma>      ::= (ryuma <roleType> [<compassDirection>] [<moves>])
<ryuo>       ::= (ryuo <roleType> [<compassDirection>] [<moves>])
<tokin>      ::= (tokin <roleType> [<compassDirection>] [<moves>])

//-------------------------------------------------------------------------------
// game.equipment.component.piece.state

<flipDisc>   ::= (flipDisc [<string>] <roleType>)
<pieceState> ::= (pieceState [<string>] <roleType> <integer> [<moves>]) |
                 <realTimePiece> | <flipDisc>

//-------------------------------------------------------------------------------
// game.equipment.component.piece.real_time

<bike>       ::= (bike <roleType> <string> [<moves>])
<realTimePiece> ::= <bike>

//-------------------------------------------------------------------------------
// game.equipment.component.piece.stratego

<stratego>   ::= <bomb> | <captain> | <colonel> | <flag> | <general> |
                 <lieutenant> | <major> | <marshal> | <miner> | <scout> |
                 <sergeant> | <spy>
<bomb>       ::= (bomb <roleType> [value:<integer>] [<moves>])
<captain>    ::= (captain <roleType> [value:<integer>] [<moves>])
<colonel>    ::= (colonel <roleType> [value:<integer>] [<moves>])
<flag>       ::= (flag <roleType> [value:<integer>] [<moves>])
<general>    ::= (general <roleType> [value:<integer>] [<moves>])
<lieutenant> ::= (lieutenant <roleType> [value:<integer>] [<moves>])
<major>      ::= (major <roleType> [value:<integer>] [<moves>])
<marshal>    ::= (marshal <roleType> [value:<integer>] [<moves>])
<miner>      ::= (miner <roleType> [value:<integer>] [<moves>])
<scout>      ::= (scout <roleType> [value:<integer>] [<moves>])
<sergeant>   ::= (sergeant <roleType> [value:<integer>] [<moves>])
<spy>        ::= (spy <roleType> [value:<integer>] [<moves>])

//-------------------------------------------------------------------------------
// game.equipment.component.piece.tafl

<tafl>       ::= <jarl> | <thrall>
<jarl>       ::= (jarl <roleType> [<moves>])
<thrall>     ::= (thrall <roleType> [<moves>])
```

```
//------------------------------------------------------------------------------
// game.equipment.component.piece.xiangqi

<xiangqi>    ::= <jiang> | <ju> | <ma> | <pao> | <shi> | <xiang> | <zu>
<xiang>      ::= (xiang <roleType> [<compassDirection>] [<moves>])
<jiang>      ::= (jiang <roleType> [<compassDirection>] [<moves>])
<ju>         ::= (ju <roleType> [<compassDirection>] [<moves>])
<ma>         ::= (ma <roleType> [<compassDirection>] [<moves>])
<pao>        ::= (pao <roleType> [<compassDirection>] [<moves>])
<shi>        ::= (shi <roleType> [<compassDirection>] [<moves>])
<zu>         ::= (zu <roleType> [<compassDirection>] [<moves>])

//------------------------------------------------------------------------------
// game.equipment.container

<container>  ::= <board> | <hand>

//------------------------------------------------------------------------------
// game.equipment.container.board

<board>      ::= (board [<string>] <shape> [<tiling>] [{<modify>}]) |
                 (board [<string>] <integer> [<integer>] [{<modify>}]) |
                 <alquerqueBoard> | <arimaaBoard> | <backgammonBoard> |
                 <boardless> | <cardBoard> | <chessBoard> |
                 <chineseCheckersBoard> | <connect4Board> | <goBoard> |
                 <graphBoard> | <halmaBoard> | <hexBoard> | <hexYBoard> |
                 <mancalaBoard> | <morrisBoard> | <peralikatumaBoard> |
                 <reversiBoard> | <senetBoard> | <shogiBoard> |
                 <snakesAndLaddersBoard> | <solitaireBoard> |
                 <stratego Board> | <surakartaBoard> | <taflBoard> |
                 <urBoard> | <wheelBoard> | <xiangqiBoard> | <puzzleBoard>
<alquerqueBoard> ::= (alquerqueBoard [<string>] <integer> <integer>)
<arimaaBoard> ::= (arimaaBoard [<string>])
<backgammonBoard> ::= (backgammonBoard [<string>])
<boardless>  ::= (boardless [<string>] <shape> <tiling>)
<cardBoard>  ::= (cardBoard [<string>])
<chessBoard> ::= (chessBoard [<string>] <integer> [<integer>])
<chineseCheckersBoard> ::= (chineseCheckersBoard [<string>])
<connect4Board> ::= (connect4Board [<string>] <integer> <integer>)
<goBoard>    ::= (goBoard [<string>] <integer>)
<graphBoard> ::= (graphBoard [<string>] <integer> <integer> [{<modify>}])
<halmaBoard> ::= (halmaBoard [<string>])
<hexBoard>   ::= (hexBoard [<string>] <integer>)
<hexYBoard>  ::= (hexYBoard [<string>] <integer>)
<mancalaBoard> ::= (mancalaBoard [<string>] <integer> <integer>)
<morrisBoard> ::= (morrisBoard [<string>] <integer> [<boolean>] [<boolean>])
<peralikatumaBoard> ::= (peralikatumaBoard [<string>] <integer>) |
                 (peralikatumaBoard [<string>] <integer> <boolean> <boolean> <
                     boolean> <boolean>)
```

```
<reversiBoard> ::= (reversiBoard [<string>] <integer>)
<senetBoard> ::= (senetBoard [<string>])
<shogiBoard> ::= (shogiBoard [<string>] <integer>)
<snakesAndLaddersBoard> ::= (snakesAndLaddersBoard [<string>] <integer> [<integer
    >])
<solitaireBoard> ::= (solitaireBoard [<string>])
<strategoBoard> ::= (strategoBoard [<string>])
<surakartaBoard> ::= (surakartaBoard [<string>] <integer> [<boolean>])
<taflBoard>  ::= (taflBoard [<string>] <integer> <boolean>)
<urBoard>    ::= (urBoard [<string>])
<wheelBoard> ::= (wheelBoard [<string>] {<integer>} [{<modify>}])
<xiangqiBoard> ::= (xiangqiBoard [<string>])

//------------------------------------------------------------------------------
// game.equipment.container.board.modify

<modify>     ::= <cut> | <joinDiago> | <joinOrtho> | <remove>
<cut>        ::= (cut int int) | (cut <modifyType> int int) | <cutAll> |
                 <cutDiago> | <cutOrtho>
<cutAll>     ::= (cutAll)
<cutDiago>   ::= (cutDiago)
<cutOrtho>   ::= (cutOrtho)
<joinDiago>  ::= (joinDiago int int)
<joinOrtho>  ::= (joinOrtho int int)
<remove>     ::= (remove int)
<modifyType> ::= Remove | Cut | CutAll | CutOrtho | CutDiago | JoinOrtho |
                 JoinDiago

//------------------------------------------------------------------------------
// game.equipment.container.puzzleBoard

<puzzleBoard> ::= (puzzleBoard [<string>] <shape> [<tiling>] [cells:<values>] [
    edges:<values>] [faces:<values>] [{<modify>}])
                 | <futoshikiBoard> | <graphPuzzleBoard> | <kakuroBoard> |
                 <lineSegmentBoard> | <nonogramBoard> | <regionPuzzleBoard> |
                 <sudokuBoard>
<dotBoard>   ::= (dotBoard [<string>] <shape> [<tiling>] [edges:<values>])
<futoshikiBoard> ::= (futoshikiBoard [<string>] <shape> {<string>} [{<string>}] [
    cells:<values>])
<graphPuzzleBoard> ::= (graphPuzzleBoard [<string>] <shape> <tiling> [vertex:<
    values>] [edges:<values>] [faces:<values>] [{<modify>}])
                 | <dotBoard> | <hashiBoard>
<hashiBoard> ::= (hashiBoard [<string>] <shape> [<tiling>] edges:<values> [{<
    modify>}])
<kakuroBoard> ::= (kakuroBoard [<string>] <shape> [{{<int>}}] [cells:<values>])
<lineSegmentBoard> ::= (lineSegmentBoard [<string>] <shape> [<tiling>] [cells:<
    values>])
<nonogramBoard> ::= (nonogramBoard [<string>] <shape> [cells:<values>])
<regionPuzzleBoard> ::= (regionPuzzleBoard [<string>] <shape> [cells:<values>] [
    cornerHints:<boolean>])
```

```
<sudokuBoard> ::= (sudokuBoard [<string>] <integer> cells:<values>)

//------------------------------------------------------------------------------
// game.equipment.container.hand

<hand>       ::= (hand [<string>] <roleType> [<integer>]) | <handDice>
<handDice>   ::= (handDice [<string>] [<integer>] <integer> [value0:<boolean>])

//------------------------------------------------------------------------------
// game.equipment.other

<map>        ::= (map {{<integer>}})
<regions>    ::= (regions [<string>] [<roleType>] {<areaType>} [<directionChoice
    >] [<string>])
                 |
                 (regions [<string>] [<roleType>] <areaType> [<directionChoice>]
                     [<string>])
                 |
                 (regions [<string>] [<roleType>] {<region>} [<string>])
                 |
                 (regions [<string>] [<roleType>] <region> [<string>])
                 | (regions [<string>] [<roleType>] {<integer>} [<string>])
<track>      ::= (track <string> <string> {<integer>} [<boolean>] [<integer>] [<
    boolean>])

//------------------------------------------------------------------------------
// game.rules

<rules>      ::= (rules [<start>] <play> [<end>])

//------------------------------------------------------------------------------
// game.rules.start

<start>      ::= (start {<startRule>}) | (start <startRule>)
<deal>       ::= (deal)
<fill>       ::= (fill <string> <region>)
<fogOfWar>   ::= (fogOfWar)
<hints>      ::= (hints [<string>] {<hint>} [<variableType>]) |
                 (hints [<string>] {{<integer>}} [<variableType>]) |
                 (hints [<string>] {<integer>} {<integer>} [<variableType>]) |
                 (hints [<string>] {{<integer>}} {<integer>} [<variableType>])
<initScore>  ::= (initScore {<roleType>} {<int>}) |
                 (initScore {<int>} {<int>}) |
                 (initScore <roleType> <int>) |
                 (initScore <int> <int>)
<place>      ::= (place <string> [<string>] [<integer>] [{<integer>}] [<region>]
    [{<string>}] [count:<integer>] [state:<integer>] [<boolean>] [hidden:{<
    boolean>}])
<placeRandomly> ::= (placeRandomly [<int>] {<string>} [<int>] [<int>] [{<boolean
    >}])
```

```
<set>         ::= (set [<variableType>] {{<integer>}})
<setCount>    ::= (setCount <integer> <integer>) |
                  (setCount <region> <integer>)
<startRule>   ::= <deal> | <fill> | <fogOfWar> | <hints> | <initScore> |
                  <place> | <placeRandomly> | <set> | <setCount>

//------------------------------------------------------------------------------
// game.rules.play

<play>        ::= (play <moves>)

//------------------------------------------------------------------------------
// game.rules.end

<end>         ::= (end <boolean> <result> [<result>]) | <byScore> |
                  <if> | <or>
<byScore>     ::= (byScore)
<if>          ::= (if <boolean> {<end>} [{<end>}]) |
                  (if <boolean> <end> [{<end>}]) |
                  (if <boolean> {<end>} [<end>]) |
                  (if <boolean> <end> [<end>])
<or>          ::= (or {<end>}) | (or <end> <end>)
<result>      ::= (result <roleType> <resultType>)
<resultType>  ::= Win | Loss | Draw | Tie | Abort

//------------------------------------------------------------------------------
// game.functions.ints

<int>         ::= int | <abs> | <add> | <column> | <count> | <counter> |
                  <directionSite> | <double> | <edge> | <forwardOnTrack> |
                  <from> | <hand> | <height> | <hint> | <if> | <indexOf> |
                  <lastEdgeMove> | <lastFromMove> | <lastToMove> | <level> |
                  <map> | <max> | <min> | <mod> | <mover> | <mult> | <next> |
                  <posPiece> | <previous> | <replayCount> | <row> | <score> |
                  <size> | <state> | <sub> | <sum> | <to> | <top> | <turn> |
                  <value> | <what> | <who>
<integer>     ::= Integer
<abs>         ::= (abs <int>)
<add>         ::= (add <int> <int>)
<column>      ::= (column <int>)
<count>       ::= (count <int> [<int>]) |
                  (count <region>)
<counter>     ::= (counter)
<directionSite> ::= (directionSite <int> [<directionChoice>])
<double>      ::= (double)
<edge>        ::= (edge)
<forwardOnTrack> ::= (forwardOnTrack <int> <int> [<int>])
<from>        ::= (from)
<hand>        ::= (hand <int> [<int>])
<height>      ::= (height <int>)
```

```
<hint>       ::= (hint)
<if>         ::= (if <boolean> <int> <int>)
<indexOf>    ::= (indexOf <regionType>) | (indexOf <roleType>) |
                 (indexOf <string> <roleType>) | (indexOf <string>)
<lastEdgeMove> ::= (lastEdgeMove [<boolean>])
<lastFromMove> ::= (lastFromMove [<boolean>])
<lastToMove> ::= (lastToMove [<boolean>])
<level>      ::= (level)
<map>        ::= (map <int>)
<max>        ::= (max <int> <int>)
<min>        ::= (min <int> <int>)
<mod>        ::= (mod <int> <int>)
<mover>      ::= (mover)
<mult>       ::= (mult <int> <int>) | (mult {<int>})
<next>       ::= (next)
<posPiece>   ::= (posPiece <string> <int>)
<previous>   ::= (previous)
<replayCount> ::= (replayCount)
<row>        ::= (row <int>)
<score>      ::= (score <int>) | (score <roleType>)
<size>       ::= (size <region>)
<state>      ::= (state <int> [<int>])
<sub>        ::= (sub <int> <int>)
<sum>        ::= (sum) | (sum {<int>}) |
                 (sum <int> <int>)
<to>         ::= (to)
<top>        ::= (top <int>)
<turn>       ::= (turn)
<value>      ::= (value <int>)
<what>       ::= (what <int> [<int>])
<who>        ::= (who <int> [<int>])

//-------------------------------------------------------------------------------
// game.functions.booleans

<le>         ::= (le <int> <int>)
<boolean>    ::= boolean | <adjacent> | <allPass> | <and> | <canMove> |
                 <configuration> | <connect> | <contains> | <double> |
                 <encircle> | <equal> | <even> | <full> | <ge> | <gt> | <if> |
                 <in> | <isCheckmate> | <isEnemy> | <isFriend> |
                 <isFriendAt> | <isMover> | <isNext> | <isPending> |
                 <isPiece> | <isPrev> | <isState> | <know> | <le> | <line> |
                 <lt> | <not> | <odd> | <or> | <allDifferent> | <connected> |
                 <count> | <crossing> | <equalParities> | <forEach> | <mult> |
                 <shape> | <solved> | <sum> | <unique> | <reachedRegion> |
                 <stalemated> | <threatened> | <visited>
<boolean>    ::= Boolean
<adjacent>   ::= (adjacent <int> <region>)
<allPass>    ::= (allPass)
<and>        ::= (and <boolean> <boolean>) |
```

```
                (and {<boolean>})
<canMove>    ::= (canMove [<moves>])
<configuration> ::= (configuration [<int>] {<integer>} [{<integer>}]) |
                (configuration [<int>] {<integer>} [<integer>])
<connect>    ::= (connect [<int>] {<region>}) |
                (connect [<int>] <roleType>)
<contains>   ::= (contains [<region>] <int>)
<double>     ::= (double)
<encircle>   ::= (encircle <int> <int>)
<equal>      ::= (equal <int> <int>)
<even>       ::= (even <int>)
<full>       ::= (full [<int>])
<ge>         ::= (ge <int> <int>)
<gt>         ::= (gt <int> <int>)
<if>         ::= (if <boolean> <boolean> [<boolean>])
<in>         ::= (in <string> [<region>]) |
                (in {<int>} <region>) |
                (in [<int>] <region>)
<isCheckmate> ::= (isCheckmate <int>)
<isEnemy>    ::= (isEnemy <int>)
<isFriend>   ::= (isFriend <int>)
<isFriendAt> ::= (isFriendAt [<int>])
<isMover>    ::= (isMover <int>)
<isNext>     ::= (isNext <int>)
<isPending>  ::= (isPending)
<isPiece>    ::= (isPiece [<int>] <int>)
<isPrev>     ::= (isPrev <int>)
<isState>    ::= (isState [<int>] <int>)
<know>       ::= (know <int> <int>)
<line>       ::= (line length:<int> [<directionChoice>] [<int>] [what:<int>] [<
    boolean>])
<lt>         ::= (lt <int> <int>)
<not>        ::= (not <boolean>)
<odd>        ::= (odd <int>)
<or>         ::= (or <boolean> <boolean>) |
                (or {<boolean>})
<reachedRegion> ::= (reachedRegion <int> <region>) |
                (reachedRegion <int> <integer>)
<stalemated> ::= (stalemated <roleType>)
<threatened> ::= (threatened <int> {<int>}) |
                (threatened <int> <int>)
<visited>    ::= (visited <int>)

//-------------------------------------------------------------------------
// game.types

<string>     ::= String
<region>     ::= Region | <hint> | <values> | <allSites> | <around> |
                <borderRegion> | <bottom> | <centre> | <column> | <corners> |
                <difference> | <directionRegion> | <edge> | <edgeFace> |
```

```
                <edgeVertex> | <empty> | <exteriors> | <facesEdge> | <hand> |
                <hintRegion> | <if> | <occupied> | <own> | <phase> |
                <playable> | <region> | <row> | <set> | <stateRegion> |
                <top> | <union> | <verticeFace> | <walk>
<areaType>    ::= Rows | Columns | AllDirections | HintRegions | Diagonals |
                SubGrids | Regions | Vertices | All
<cardType>    ::= JR | JB | JW | C2 | C3 | C4 | C5 | C6 | C7 | C8 | C9 | C10 |
                CJ | CQ | CK | CA | D2 | D3 | D4 | D5 | D6 | D7 | D8 | D9 |
                D10 | DJ | DQ | DK | DA | H2 | H3 | H4 | H5 | H6 | H7 | H8 |
                H9 | H10 | HJ | HQ | HK | HA | S2 | S3 | S4 | S5 | S6 | S7 |
                S8 | S9 | S10 | SJ | SQ | SK | SA
<regionType> ::= Empty | NotEmpty | Own | NotOwn | Enemy | NotEnemy | All | None
<roleType>    ::= None | P1 | P2 | P3 | P4 | P5 | P6 | P7 | P8 | Any | Each |
                All | Mover | Next | Prev | Odd | Even | NonMover |
                Opposite | Empty | Own | Enemy | Ally | NonAlly | Partner |
                NonPartner
<shapeType>   ::= Square | Rectangle | Triangle | Hexagone
<stepType>    ::= F | B | L | R
<variableType> ::= Vertex | Edge | Face | Hint | Region
```

References

Auer, P., Cesa-Bianchi, N., & Fischer, P. (2002). Finite-time Analysis of the Multiarmed Bandit Problem. *Machine Learning*, *47*(2–3), 235–256.

Browne, C. (2016). A Class Grammar for General Games. *Computers and Games*, *LNCS 10068*, 167–182.

Browne, C., Powley, E., Whitehouse, D., Lucas, S., Cowling, P. I., Rohlfshagen, P., ... Colton, S. (2012). A Survey of Monte Carlo Tree Search Methods. *IEEE Transactions on Computational Intelligence and AI in Games*, *4*(1), 1–49.

Browne, C., Soemers, D. J. N. J., & Piette, E. (2019). Strategic features for general games. In *Proceedings of the 2nd workshop on knowledge extraction from games (keg)* (pp. 70–75).

Cazenave, T. (2015). Generalized Rapid Action Value Estimation. In Q. Yang & M. Woolridge (Eds.), *Proceedings of the Twenty-Fourth International Joint Conference on Artificial Intelligence (IJCAI 2015)* (pp. 754–760). AAAI Press.

Perez, D., Dieskau, J., Hünermund, M., Mostaghim, S., & Lucas, S. M. (2015). Open Loop Search for General Video Game Playing. In *Proceedings of the Genetic and Evolutionary Computation Conference* (pp. 337–344).

Silver, D., Schrittwieser, J., Simonyan, K., Antonoglou, I., Huang, A., Guez, A., ... Hassabis, D. (2017). Mastering the game of Go without human knowledge. *Nature*, *550*, 354–359.

Soemers, D. J. N. J., Piette, É., & Browne, C. (2019). Biasing MCTS with features for general games. In *Proceedings of the 2019 ieee congress on evolutionary computation (cec 2019)* (p. 442-449).

Stephenson, M., Piette, É., Soemers, D. J. N. J., & Browne, C. (2019). Ludii as a Competition Platform. In *Proceedings of the IEEE Conference on Games (COG 2019).*