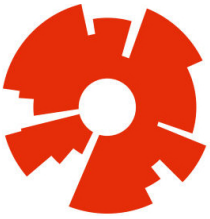


HÁSKÓLINN Í REYKJAVÍK
REYKJAVIK UNIVERSITY

Abstract Board Games
A Framework and a Language

Prepared By:
Bjarni Dagur Thor Kárason (bjarnidt19@ru.is)
Guðmundur Freyr Ellertsson (gudmundurfe19@ru.is)

Supervisor:
Prof. Yngvi Björnsson (yngvi@ru.is)

Examiner:
Dr. Henning Arnór Úlfarsson (henningu@ru.is)

T-404-LOKA
BSc Computer Science
Reykjavik University, Department of Computer Science
May, 2022

Acknowledgements

The authors would like to thank professor Yngvi Björnsson for supervising this final project and guiding the authors through the vast world of general game playing.

Contents

1 Introduction **7**

2 Related work **8**

3 Background **8**
3.1 General Game Playing . . . 8
3.2 Search and Evaluation . . . 8
3.2.1 Game Trees . . . 9
3.2.2 Evaluating Game States . . . 9
3.2.3 Minimax Search . . . 10
3.2.4 Monte-Carlo Tree Search . . . 10
3.3 Breakthrough . . . 11
3.4 Parsing . . . 11

4 The Abstract Boardgames Description Language **14**
4.1 Writing Abstract Boardgames Descriptions . . . 15
4.2 Methods . . . 15
4.2.1 Storing and Calling Variables, Predicates, Move Effects, and Terminal Conditions 15
4.2.2 Parsing Regular Expressions and Resolving Macro Calls . . . 16
4.2.3 Parsing Game Descriptions . . . 16
4.2.4 Generating Legal Moves . . . 18
4.3 Optimizations . . . 20
4.4 Experiments . . . 24
4.5 Towards a Self-Contained Language . . . 25

5 The Game Playing Agent **26**
5.1 Overview . . . 27
5.2 Domain Knowledge . . . 27
5.3 Architecture . . . 27
5.4 Creating Training Data . . . 29
5.5 Experiments . . . 29

6 A General Game Graphical User Interface **31**

7 Conclusion and Discussion **31**

8 Future Work **33**
8.1 Optimizations . . . 33
8.2 A Self-Contained Language . . . 33

References **34**

Appendices **36**

A Abstract Boardgames Grammar **36**

B C++ ABG API **37**

C Game Descriptions in ABG **38**
C.1 Breakthrough 38
C.2 Chess 38
C.3 Connect 4 40
C.4 Tic-Tac-Toe 40

D Self-contained Abstract Boardgames Grammar **43**

List of Acronyms

ABG Abstract Boardgames

AI Artificial Intelligence

AST Abstract Syntax Tree

CFG Context-Free Grammar

DFA Deterministic Finite Automaton

EBNF Extended Backus-Naur Form

GDL Game Description Language

GGP General Game Playing

GUI Graphical User Interface

LL Left-to-right, Leftmost derivation

MCTS Monte-Carlo Tree Search

NFA Nondeterministic Finite Automaton

RAM Random Access Memory

RBG Regular Boardgames

UCT Upper Confidence Tree

List of Figures

1 The first two levels of the game tree for Tic-tac-toe, up to symmetry (from Wikipedia). 9
2 Minimax tree search example (from Wikipedia). 10
3 MCTS search example (from Wikipedia). 12
4 8x8 Breakthrough initial state. 13
5 AST example (from Yngvi Björnsson's Compilers course). 13
6 Breakthrough in ABG. 16
7 Breakthrough in GDL (from games.ggp.org). 17
8 Breakthrough in RBG (from RBG GitHub). 18
9 Breakthrough in Ludii (from ludii.games). 19
10 Resolving a macro call in a rule. 19
11 Resolving a macro call in a macro. 20
12 NFA to generate moves for a white pawn in chess. 21
13 DFA to generate moves for a white pawn in chess. 22
14 Minimized DFA to generate moves for a white pawn in chess. 23
15 Part of a call graph profile of the first language implementation. 23
16 Part of a call graph profile after implementing anti-effects. 24
17 Example 3x3 Breakthrough state. 28
18 Complete neural network architecture. 30
19 Convolutional block architecture. 30
20 Residual block architecture. 30
21 Value head architecture. 30
22 4x4 Breakthrough initial state. 31
23 Chess in our GUI. 32
24 Tic-tac-toe in our GUI. 32
25 Breakthrough in Abstract Boardgames (ABG). 39
26 Chess in ABG (more readable on GitHub). 41
27 Connect 4 in ABG. 42
28 Tic-Tac-Toe in ABG. 42

List of Tables

1 Efficiency comparison between ABG, Ludii, and RBG using the perft metric, measured in states/second (higher is better). 24
2 Efficiency comparison between ABG, Ludii, and RBG using the flat Monte-Carlo metric, measured in states/second (higher is better). 25
3 The state's (Fig 17) representation. 28
4 Evaluation of agents as games won, drawn, and lost from the agent's perspective. 29
5 Abstract Boardgames EBNF-style grammar 36
6 Self-contained Abstract Boardgames EBNF-style grammar 44

1 Introduction

Board games have been a part of humanity for thousands of years [1]. Ever since Alan Turing considered the question, "can machines think?" in a seminal paper published in 1950 [2], games have played a role in the development of intelligent machines. Over the years, Artificial Intelligence (AI) researchers have developed game-playing systems capable of playing games of various complexity at an expert level [3] [4] [5]. However, these game-playing systems were all made to play a single game and relied heavily on human expert knowledge. In a way, the system's intelligence lies within its creators, not the system itself.

The goal of General Game Playing (GGP) is to create game-playing systems that can play a wide variety of games at a high level, given only the descriptions of the rules of the game. Such systems cannot rely on human knowledge. The system's intelligence should lie within itself. A recent example of such a system is *AlphaZero* [6], which learned to play Go, chess, and shogi at an expert level from scratch.

To create a GGP system, we need a way to describe the rules of a game. Several GGP languages exist that do exactly that [7] [8] [9], each with its strengths and weaknesses concerning efficiency, simplicity, and formalism. Game Description Language (GDL) is expressive, but reasoning is slow. Regular Boardgames (RBG) allows for fast reasoning but describes everything about a game using regular expressions, leading to unintuitive game descriptions. Ludii allows for fast reasoning as well, but is more like a framework than a language, as every *ludeme* (the building blocks of Ludii game descriptions) is implemented in Java.

In this report, we introduce *Abstract Boardgames* (ABG), a well-defined general game description language, combining new ideas with good ideas from existing languages. Its main goal is to be expressive enough to describe the rules of popular chess-like games, yet simple enough to allow effective computation of complex games while allowing for easy to create and understandable game descriptions. The language is based on regular expressions to describe piece movements, and user-defined game concepts to describe when a move is legal, how a move affects the game state, and the game's terminal conditions.

This report is structured as follows. In the next section, we discuss related work in more detail. Then we go over the necessary prerequisites for this report. Thereafter we describe our Abstract Boardgames language and compare its reasoning efficiency against other GGP languages. Next, we showcase a proof-of-concept intelligent agent that uses our language to describe games which it subsequently learns to play at an intermediate-expert level. We also demonstrate a rudimentary user interface to play games against humans or artificial agents. Finally, we discuss the conclusions of the project and possible future work.

The source code of this project is published on GitHub[1] under the GNU General Public License, version 2.

[1] `https://github.com/bjarnithor99/abstractboardgames`

2 Related work

Several GGP languages have been proposed over the years, most notably GDL [7], and later Regular Boardgames [8] and Ludii [9]. GGP languages are used to describe the game itself, not a game strategy. This includes the initial state of the game, the players, their pieces, and how they can be moved, as well as the effects a move has on the game's state, when the game ends, and who is the winner.

GDL is a first-order-logic language based on Datalog, a syntactic subset of Prolog. It can describe any turn-based, finite, deterministic, n-player game with perfect information. Extensions of GDL have been developed to support games with imperfect and incomplete information [10], and even games with epistemic goals [11]. However, logic resolution is computationally expensive, making many games described in GDL infeasible to learn to play at a decent level. Another downside of GDL is that every game concept has to be defined from scratch, making game descriptions long and unintuitive.

Regular Boardgames is a language based on regular expressions. It aims to "allow effective computation of complex games, while at the same time being universal and allowing concise and easy to process game descriptions that intuitively correspond to the game structure" and has been shown to be universal for the class of finite deterministic games with perfect information, and more efficient than GDL [8]. However, as a game described in RBG is described completely using regular expressions, the description can be non-trivial to create and understand.

The Ludii general game system describes games using a broad set of predefined *ludemes*, high-level conceptual elements of a game, allowing for concise and human-understandable game descriptions. As ludemes are implemented in Java, Ludii can theoretically support any game concept programmable in Java. Ludii has been shown to be universal for the class of finite deterministic games with perfect information, more efficient than GDL, and competitive in terms of performance with RBG [9]. However, as Ludii relies on programmable game concepts, it lacks the mathematical formalism offered by both GDL and RBG.

Both RBG and Ludii are efficient tools for AI researchers and practitioners.

3 Background

3.1 General Game Playing

Instead of designing an AI system to play one particular game, the goal of GGP is to design an AI system that given the rules of a previously unknown game, learns to play that game successfully. Therefore, general game players cannot rely on game-specific knowledge or evaluation functions as many successful specialized game players, such as Deep Blue [4], have done.

To design a general game player, we need a way to formally describe a game, which is often done using a game description language. The most commonly used description language is GDL [7], which can describe any turned-based, finite, deterministic, n-player game with perfect information. Other languages such as RBG [8] and Ludii [9] describe a narrower set of games more efficiently [12]. See Section 2 for discussion on game description languages related to this project.

3.2 Search and Evaluation

As GGP agents learn previously unknown games without human intervention, they need a way to search the game tree and evaluate game states in order to find good moves. The first successful GGP agents used a minimax-based tree search with an automatically learned heuristic function [13] [14]. Later agents used Monte-Carlo Tree Search (MCTS) instead of minimax search [15]. As deep learning became more popular, GGP agents began using deep neural networks to evaluate game states, and guide the MCTS [6].

3.2.1 Game Trees

A game tree is a directed graph where each node represents a possible game state, and edges between nodes represent moves. Two nodes u, v are connected if there exists a move that if played in game state u results in game state v. The root of the tree is the game's initial state, and the leaf nodes are terminal states. A part of the game tree for Tic-tac-toe is shown in Figure 1.

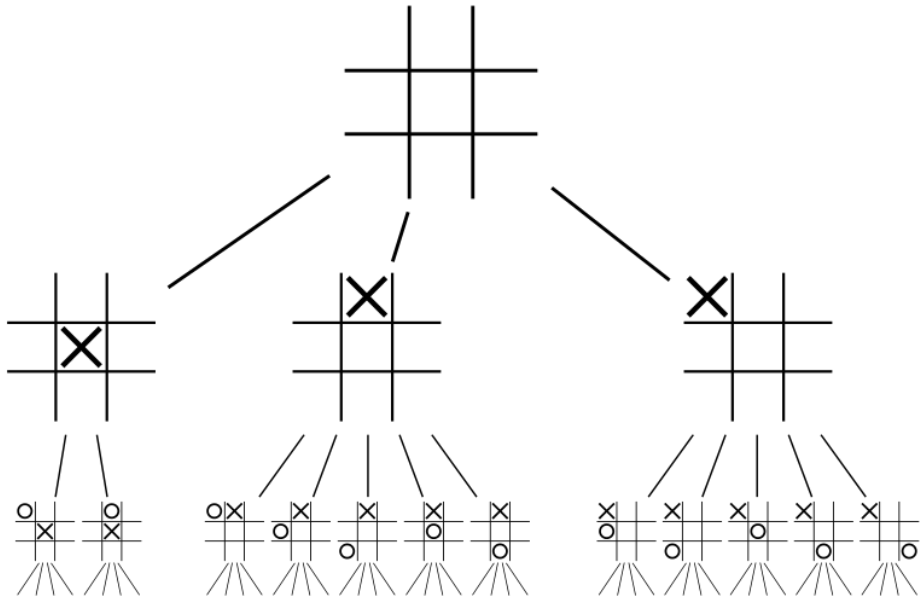


Figure 1: The first two levels of the game tree for Tic-tac-toe, up to symmetry (from Wikipedia).

As game trees for most games are too large to search exhaustively (the game tree for a simple game like Tic-tac-toe has 255,168 leaf nodes [16]), successful game playing agents need a way to search the tree efficiently. One optimization is to prune symmetrically equivalent game states. For example, in Tic-tac-toe, the first player can mark one of nine squares, but there are only three distinct successor states, as is illustrated in Figure 1.

We will discuss two commonly used tree search algorithms, minimax-based search, and MCTS.

3.2.2 Evaluating Game States

As it is infeasible to completely search the game tree for most games, we need to be able to assign a "goodness score" to arbitrary game states without relying on the leaf nodes of the tree. The "goodness score" of a game state approximates the expected outcome of the game if it were played from the current game state.

Strong specialized game players (like Stockfish for chess) often use carefully handcrafted evaluation functions with domain-specific features. However, GGP agents cannot rely on domain-specific knowledge. Recently, GGP agents have begun using deep neural networks to evaluate game states. A well-known example is AlphaZero [6], a computer agent that learned to play chess, shogi, and Go, at a superhuman level entirely from self-play and without any domain-specific knowledge.

3.2.3 Minimax Search

Minimax is a recursive algorithm for choosing the next optimal move in an n-player game [17]. For the purposes of this report, we are mostly interested in two-player games. Assume we have two players, *black* and *white*. Minimax finds the next move by working up from the leaf nodes of the game tree. The idea is that a win for *black* is a loss for *white*, and if *white* can pick between two moves, one of which leads to a guaranteed win for *black* and the other to a draw at best, then the latter move is better for *white*. It is assumed that both players will play optimally, that is, at each game state a player will pick the move that maximizes their chances of winning (or equivalently, minimize chances of losing).

Usually, a value is assigned to each game state, say $-\infty$ if *black* is guaranteed to win, $+\infty$ if *white* is guaranteed to win, and a value according to some evaluation function otherwise. The value of a move from *white*'s perspective is the maximum value of the states resulting from *black*'s possible replies. Conversely, the value of a move from *black*'s perspective is the minimum value of the states resulting from *white*'s possible replies. For this reason, *white* is called the maximizing player, *black* is called the minimizing player, and the algorithm is called minimax. An example minimax search tree is shown in Figure 2.

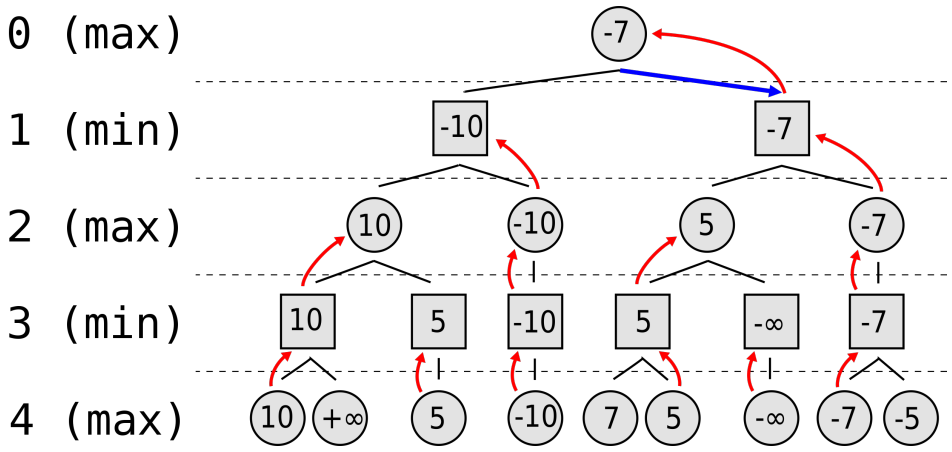


Figure 2: Minimax tree search example (from Wikipedia).

A common optimization for minimax search is alpha-beta pruning [18]. The alpha-beta pruning algorithm stops evaluating a move once at least one possible reply has been found that proves that the move is worse than a previously searched move. Such moves do not have to be searched further. Depending on the order in which moves are searched, alpha-beta pruning can search the game tree up to twice as deep with the same amount of computation as minimax search [18].

3.2.4 Monte-Carlo Tree Search

Instead of fully searching the game tree up to a certain depth like minimax search, MCTS samples the game tree. Traditionally, MCTS uses a Upper Confidence Tree (UCT) score [19] to focus on analyzing the most promising moves, while also exploring other moves. To choose the next move, MCTS first runs multiple rounds of search, typically until a timer runs out. During the search, selecting the most promising move is called *exploitation*, while selecting other moves is called *exploration*. Each round consists of four steps [20]:

- *Selection*: Start from the root of the search tree and select a child node until a leaf node L is reached. The root node is the current game state and a leaf node is a game state from which no

simulations have been run. Child nodes are typically selected using some aforementioned UCT score to balance exploitation and exploration.

- *Expansion*: Unless L is a terminal state (win/loss/draw), the node is expanded. To expand L, a child node is generated for every game state resulting of a legal move made in L. Once L is expanded, a child node C is chosen.
- *Simulation*: Do a rollout from C, that is, play the game from C until a terminal state is reached. Typically, random moves are chosen during the rollout.
- *Backpropagation*: Each node in the search tree keeps track of often it has been visited, and how often it has been on a winning path. Use the result from the rollout to update the nodes on the path from the root to C. If the current player won the rollout, then increment both the visited and winning counters of each node in the path by one. Otherwise, just increment the visited counter.

Once all the search rounds have been run, the child of the root node that was visited most is considered to be the most promising game state, and the corresponding move is chosen as the next move. One search round is shown as an example in Figure 3.

3.3 Breakthrough

Throughout this report, we often use Breakthrough [21] in examples. Breakthrough is a two-player turn-taking board game, played on an 8x8 board, set up as is shown in Figure 4. Each player moves one piece per turn. Pieces move one square straight or diagonally forward, as shown in Figure 4 using green arrows. Pieces can capture by moving diagonally forward. The first player to reach the opponent's edge of the board wins. A player can also win by capturing all of the opponent's pieces.

3.4 Parsing

To be able to formalize board games in our own language, and then create GGP agents that learn to play the game using only the game description, we need structured information about the game and a way to extract the information.

By structured information, we mean text written in a language defined by some grammar. For now, assume a Context-Free Grammar (CFG). To extract structured information we use a parser for the language the information is written in. First, a *lexer* performs *lexical-analysis* on the source. Its job is to convert the source from a sequence of characters into a sequence of *tokens* (also known as *lexemes*) that represent meaningful symbols in the source language [22]. This process is sometimes referred to as *tokenization*. Next, a *parser* performs *syntactical-analysis*. Its job is to verify that the stream of tokens produced by the lexer adheres to the syntactical rules of the source language, and to match the token stream against the language's grammar to create an Abstract Syntax Tree (AST), a tree representation of the input [23]. As an example, a grammar describing simplified arithmetic expressions that allow only addition, subtraction, and multiplication of numbers, an expression conforming to the grammar, and its AST is shown in Figure 5.

To structure information about a board game we need a formal language defined by some grammar, that can describe a game and its rules. For the purposes of this report, we are interested in Left-to-right, Leftmost derivation (LL) grammars, specifically LL(1) grammars. A LL(n) grammar is a CFG grammar that can be parsed by reading the input token stream from left-to-right, producing a left-to-right-derivation, using at most n token look-ahead [24]. LL(1) grammars are of particular interest due to efficiency reasons [25].

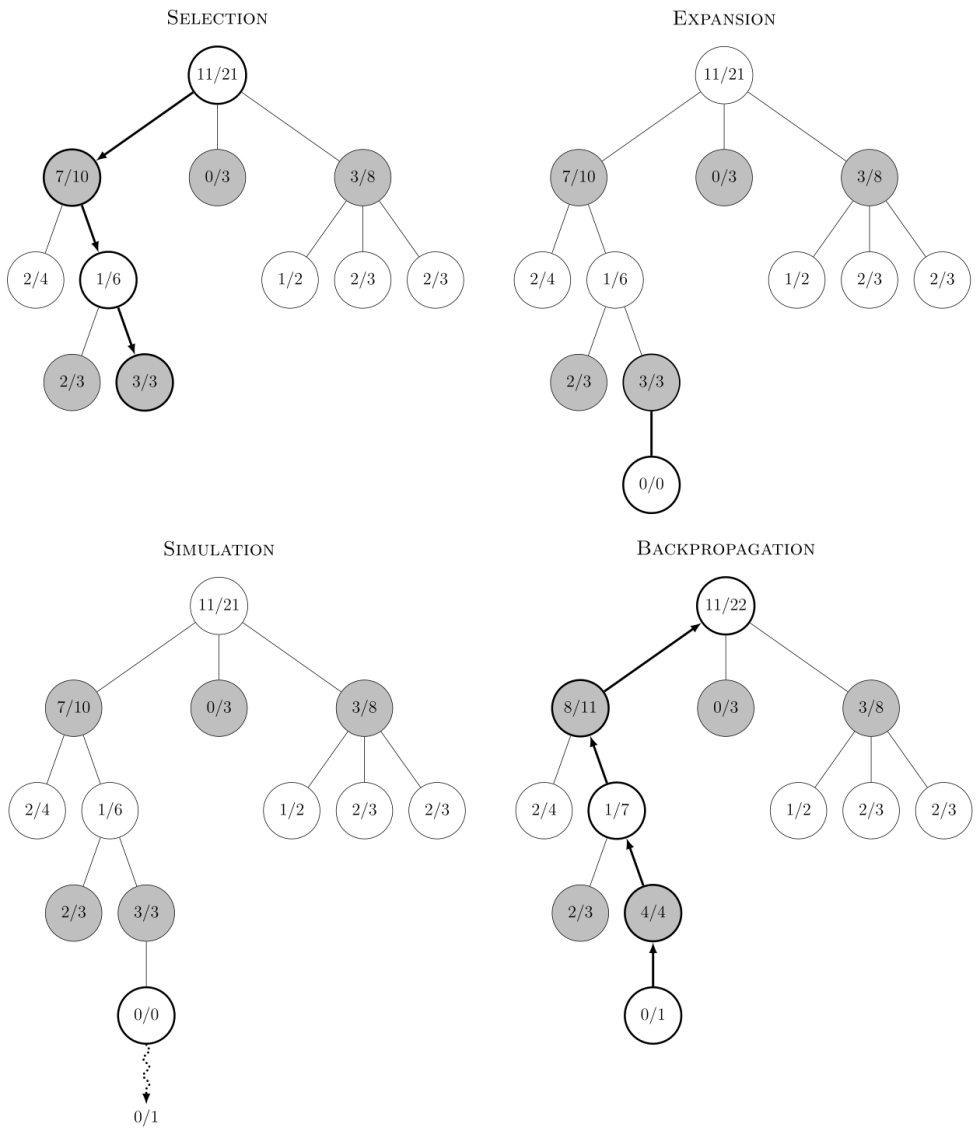


Figure 3: MCTS search example (from Wikipedia).

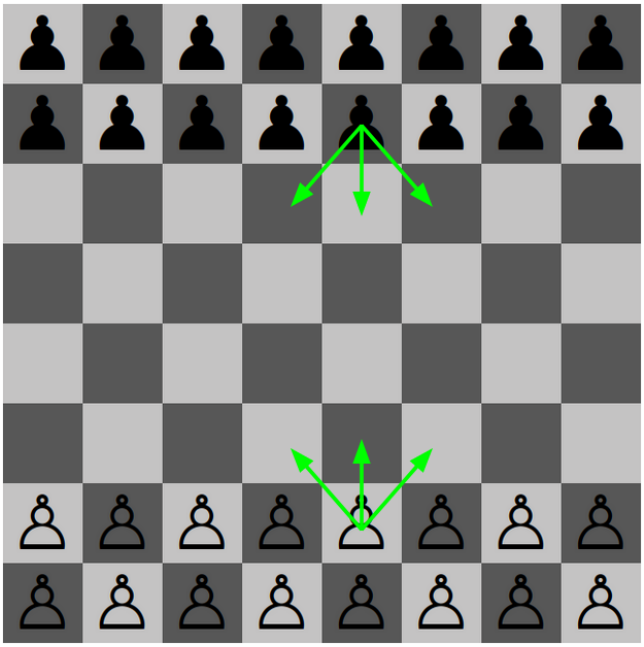


Figure 4: 8x8 Breakthrough initial state.

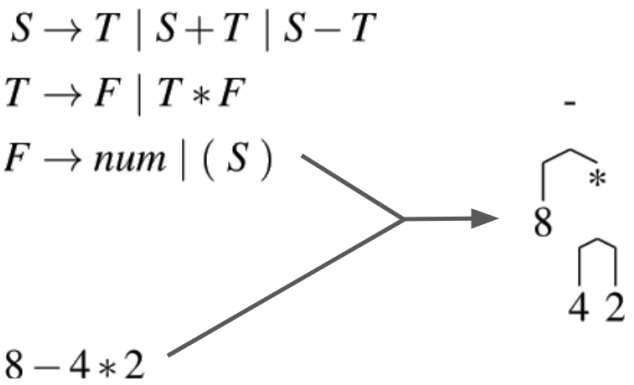


Figure 5: AST example (from Yngvi Björnsson's Compilers course).

4 The Abstract Boardgames Description Language

The Abstract Boardgames language can describe turn-based, finite, deterministic, n-player games with perfect information. An Abstract Boardgames game description is a 5-tuple

$$\mathcal{G} = \{Players, Pieces, Board, Rules, PostConditions\}$$

which together with user-defined variables, predicates, move effects, and terminal conditions, constitutes a complete description of a board game. The variables, predicates, move effects, and terminal conditions, are defined separately from the game description in C++.

Players is a finite non-empty set of players participating in the game. As an example, in Breakthrough we have *Players = {black, white}*.

Pieces is a finite non-empty set of pieces that can be placed on the game board. Each piece is owned by zero-or-more players. A special empty piece may be defined to indicate empty squares. In their turn, players can move only the pieces they own. If a piece moves to an occupied square it replaces the existing piece as we only allow at most one piece on each square. As an example, in Breakthrough we have *Pieces = {empty, bPawn(black), wPawn(white)}*.

Board is a NxM array representing the initial state of the game board. Each element in the array is a piece. As an example, in Breakthrough, we have an 8x8 board

$$Board = \begin{bmatrix} bPawn & bPawn & \dots & bPawn \\ bPawn & bPawn & \dots & bPawn \\ empty & empty & \dots & empty \\ \dots & & & \\ empty & empty & \dots & empty \\ wPawn & wPawn & \dots & wPawn \\ wPawn & wPawn & \dots & wPawn \end{bmatrix}$$

Rules is a set of rules for the pieces that are owned by at least one player, one for each piece. Each rule is given as a regular expression whose input symbols we call *letters*. A letter $[\Delta x, \Delta y, \text{predicate}]\{\text{effect}\}$ describes a part of a piece's move. The $\Delta x, \Delta y$ references another square relative to the piece's current square, and the predicate evaluates to true if the movement is legal. Finally, effect describes the effect the move has on the game state. It is optional to specify an effect, in which case a default effect will be used. As an example, the rule $wPawn = [0, 1, Empty]$ says that $wPawn$ can move one step forward along the y-axis if the destination square is empty, and the move has the default effect. The rule $wPawn = [0, 1, TopRow]\{PromoteToQueen\}$ says that $wPawn$ can move one step forward along the y-axis if the destination square is in the top row of the game board, and be promoted to a queen as an effect (applicable in chess).

Letters can be composed using common regular expression operators such as: concatenation $\cdot$ (the dot is usually omitted), Kleene star $*$ (zero-or-more occurrences), the question mark ? (zero-or-one occurrence), the plus + (one-or-more occurrences), and the or operator | (alternation). As an example, the following rule describes all possible moves for $wPawn$ in Breakthrough:

$$wPawn = [0, 1, Empty] \mid [-1, 1, Empty] \mid [1, 1, Empty] \mid [-1, 1, Opponent] \mid [1, 1, Opponent]$$

Concatenated letters are applied in succession. For example, the rule

$$wPawn = [0, 1, Empty][0, 1, Opponent]$$

says that $wPawn$ can move one forward along the y-axis to an empty square, and then one more forward to a square occupied by the opponent.

PostConditions is a set of conditions that must hold after a player makes a move. Conditions are defined for player/piece pairs and are given as regular expressions composed of letters without side effects. A postcondition holds if the regular expression is not matched. After a player makes a move, all its postconditions are checked for each of the pieces they are defined for. For example, the postcondition

$$white\ wKing = [-1, 1, BPawn] \mid [1, 1, BPawn]$$

says that after the *white* player makes a move, none of its *wKing* pieces can be attacked by *bPawn* (the *BPawn* predicate returns true only for squares occupied by *bPawn*).
Variables can be used in predicates, move effects, and terminal conditions, to express complex behavior that is difficult to represent using only regular expressions. Predicates should not alter the game state at all (including variables), but move effects and terminal conditions can, for example, to indicate that a pawn can be captured *en passant*, or set the players' final scores.
Predicates are a part of letters and restrict when the corresponding move is legal. Predicates can read the game state but not alter it.
Move effects are a part of letters and describe the effect a move has on the game state. Move effects can be omitted from letters, in which case a default effect is assumed. Move effects can read and alter the game state.
Terminal conditions describe when the game should be terminated, and are checked every time a player makes a move, or if a player has no legal moves. Once a terminal condition is satisfied, the game is terminated. Terminal conditions can read the game state and set the players' final scores.

4.1 Writing Abstract Boardgames Descriptions

Our syntax of the Abstract Boardgames language is expressed in Extended Backus-Naur Form (EBNF)-style grammar, given in Appendix A, and corresponds to the abstract definition above. The syntax has a simple function-style macro substitution system to allow more concise and human-readable game descriptions. A macro can have any number of parameters, and a macro can call other macros, but only the Δx and Δy part of letters can be parameterized. This restriction is to avoid complicated macro chains.

A complete game description of the game Breakthrough in Abstract Boardgames is shown in Figure 6. A complete description of Breakthrough in GDL, RBG, and Ludii are shown in Figures 7, 8 and 9 respectively, for comparison.

4.2 Methods

In this section we describe various interesting and important Abstract Boardgames implementation details.

4.2.1 Storing and Calling Variables, Predicates, Move Effects, and Terminal Conditions

As mentioned before, the Abstract Boardgames language is supplemented with variables, predicates, move effects, and terminal conditions written in C++. All the user-defined variables are members of a special `Variables` class owned by the game environment. As predicates, move effects, and terminal conditions can access the game environment, they can also access the environment's `Variables` class.

Predicates, move effects, and terminal conditions are implemented as functors. Each functor is mapped to a name that can be referenced in Abstract Boardgames descriptions.

As it is relatively expensive to copy memory, each move effect also implements a "anti-effect", which can reverse the changes made to the game state by the move effect. Instead of copying and storing

```
players = white, black;

pieces = empty, wPawn(white),
                bPawn(black);

macro forward(dy) = [0, dy, Empty] | [-1, dy, Empty] | [1, dy, Empty];
macro capture(dy) = [-1, dy, Opponent] | [1, dy, Opponent];

rule bPawn = forward(-1) | capture(-1);
rule wPawn = forward( 1) | capture( 1);

board_size = 8, 8;
board = bPawn, bPawn,  bPawn,  bPawn,  bPawn, bPawn,  bPawn,  bPawn,
        bPawn, bPawn,  bPawn,  bPawn,  bPawn, bPawn,  bPawn,  bPawn,
        empty, empty,  empty,  empty,  empty, empty,  empty,  empty,
        empty, empty,  empty,  empty,  empty, empty,  empty,  empty,
        empty, empty,  empty,  empty,  empty, empty,  empty,  empty,
        empty, empty,  empty,  empty,  empty, empty,  empty,  empty,
        wPawn, wPawn,  wPawn,  wPawn,  wPawn, wPawn,  wPawn,  wPawn,
        wPawn, wPawn,  wPawn,  wPawn,  wPawn, wPawn,  wPawn,  wPawn;
```

Figure 6: Breakthrough in ABG.

the whole game state before executing a move effect, only the changes are stored. More optimizations are discussed in Section 4.3.

4.2.2 Parsing Regular Expressions and Resolving Macro Calls

In Abstract Boardgames descriptions, macros, piece rules, and postconditions are given as regular expressions. Each regular expression in an Abstract Boardgames description is parsed and an AST that represents it is created. If a piece rule calls a macro, then the macro's AST is added to the rule's AST at the correct location with the parameterized values replaced with the desired values. If a macro calls another macro, then the AST of the callee is added to the AST of the caller, and the parameterized values are adjusted accordingly. Two graphical examples are shown in Figures 10 and 11.

Using the visitor design pattern, an Nondeterministic Finite Automaton (NFA) is created from each regular expression AST. Then the NFA is translated to an equivalent Deterministic Finite Automaton (DFA), which is then minimized. We do this since it is more convenient to generate moves for a piece using a state machine compared to its corresponding regular expression. We note that regular expressions, NFAs, and DFAs are computationally equivalent [26], but translating an NFA to a DFA can cause an exponential blowup in the number of states [27]. However, we have not observed this to be a problem. In Figures 12, 13, and 14 we show an NFA created from a regular expression describing all the possible moves for a white pawn in chess, an equivalent DFA, and the minimized DFA respectively. The figures also illustrate that a large NFA can be translated into a large DFA, but that does not guarantee that the minimized DFA will be large as well.

4.2.3 Parsing Game Descriptions

As Abstract Boardgames's grammar (see Table 5 in Appendix A) is a LL(1) grammar, we can parse game descriptions effectively using a recursive-descent parser without any backtracking. To provide better error messages and ensure that no undefined players/pieces are used in game descriptions, the parser enforces a certain ordering of definitions:

```
(role white)
(role black)

(<= (base (cellHolds ?x ?y ?p))
    (index ?x)
    (index ?y)
    (role ?p))
(<= (base (control ?p))
    (role ?p))

(<= (input ?p noop)
    (role ?p))
(<= (input white (move ?x ?y1 ?x ?y2))
    (index ?x)
    (succ ?y1 ?y2))
(<= (input white (move ?x1 ?y1 ?x2 ?y2))
    (succ ?y1 ?y2)
    (succ ?x1 ?x2))
(<= (input white (move ?x1 ?y1 ?x2 ?y2))
    (succ ?y1 ?y2)
    (succ ?x2 ?x1))
(<= (input black (move ?x ?y1 ?x ?y2))
    (index ?x)
    (succ ?y2 ?y1))
(<= (input black (move ?x1 ?y1 ?x2 ?y2))
    (succ ?y2 ?y1)
    (succ ?x1 ?x2))
(<= (input black (move ?x1 ?y1 ?x2 ?y2))
    (succ ?y2 ?y1)
    (succ ?x2 ?x1))

(init (cellHolds 1 1 white))
(init (cellHolds 2 1 white))
(init (cellHolds 3 1 white))
(init (cellHolds 4 1 white))
(init (cellHolds 5 1 white))
(init (cellHolds 6 1 white))
(init (cellHolds 7 1 white))
(init (cellHolds 8 1 white))
(init (cellHolds 1 2 white))
(init (cellHolds 2 2 white))
(init (cellHolds 3 2 white))
(init (cellHolds 4 2 white))
(init (cellHolds 5 2 white))
(init (cellHolds 6 2 white))
(init (cellHolds 7 2 white))
(init (cellHolds 8 2 white))

(init (cellHolds 1 7 black))
(init (cellHolds 2 7 black))
(init (cellHolds 3 7 black))
(init (cellHolds 4 7 black))
(init (cellHolds 5 7 black))
(init (cellHolds 6 7 black))
(init (cellHolds 7 7 black))
(init (cellHolds 8 7 black))
(init (cellHolds 1 8 black))
(init (cellHolds 2 8 black))
(init (cellHolds 3 8 black))
(init (cellHolds 4 8 black))
(init (cellHolds 5 8 black))
(init (cellHolds 6 8 black))
(init (cellHolds 7 8 black))
(init (cellHolds 8 8 black))

(<= (legal white (move ?x ?y1 ?x ?y2))
    (true (control white))
    (true (cellHolds ?x ?y1 white))
    (succ ?y1 ?y2)
    (cellEmpty ?x ?y2))
(<= (legal white (move ?x1 ?y1 ?x2 ?y2))
    (true (control white))
    (true (cellHolds ?x1 ?y1 white))
    (succ ?y1 ?y2)
    (succ ?x1 ?x2)
    (not (true (cellHolds ?x2 ?y2 white))))
(<= (legal white (move ?x1 ?y1 ?x2 ?y2))
    (true (control white))
    (true (cellHolds ?x1 ?y1 white))
    (succ ?y1 ?y2)
    (succ ?x2 ?x1)
    (not (true (cellHolds ?x2 ?y2 white))))

(<= (legal black (move ?x ?y1 ?x ?y2))
    (true (control black))
    (true (cellHolds ?x ?y1 black))
    (succ ?y2 ?y1)
    (cellEmpty ?x ?y2))
(<= (legal black (move ?x1 ?y1 ?x2 ?y2))
    (true (control black))
    (true (cellHolds ?x1 ?y1 black))
    (succ ?y2 ?y1)
    (succ ?x1 ?x2)
    (not (true (cellHolds ?x2 ?y2 black))))
(<= (legal black (move ?x1 ?y1 ?x2 ?y2))
    (true (control black))
    (true (cellHolds ?x1 ?y1 black))
    (succ ?y2 ?y1)
    (succ ?x2 ?x1)
    (not (true (cellHolds ?x2 ?y2 black))))

(<= (legal white noop)
    (true (control black)))
(<= (legal black noop)
    (true (control white)))

(<= (next (cellHolds ?x2 ?y2 ?player))
    (role ?player)
    (does ?player (move ?x1 ?y1 ?x2 ?y2)))
(<= (next (cellHolds ?x3 ?y3 ?state))
    (true (cellHolds ?x3 ?y3 ?state))
    (role ?player)
    (does ?player (move ?x1 ?y1 ?x2 ?y2))
    (distinctCell ?x1 ?y1 ?x3 ?y3)
    (distinctCell ?x2 ?y2 ?x3 ?y3))

(<= (next (control white))
    (true (control black)))
(<= (next (control black))
    (true (control white)))

(<= terminal
    whiteWin)
(<= terminal
    blackWin)

(<= (goal white 100)
    whiteWin)
(<= (goal white 0)
    (not whiteWin))

(<= (goal black 100)
    blackWin)
(<= (goal black 0)
    (not blackWin))

(<= (cell ?x ?y)
    (index ?x)
    (index ?y))

(<= (cellEmpty ?x ?y)
    (cell ?x ?y)
    (not (true (cellHolds ?x ?y white)))
    (not (true (cellHolds ?x ?y black))))

(<= (distinctCell ?x1 ?y1 ?x2 ?y2)
    (cell ?x1 ?y1)
    (cell ?x2 ?y2)
    (distinct ?x1 ?x2))
(<= (distinctCell ?x1 ?y1 ?x2 ?y2)
    (cell ?x1 ?y1)
    (cell ?x2 ?y2)
    (distinct ?y1 ?y2))

(<= whiteWin
    (index ?x)
    (true (cellHolds ?x 8 white)))
(<= blackWin
    (index ?x)
    (true (cellHolds ?x 1 black)))


(<= whiteWin
        (not blackCell))
(<= blackWin
        (not whiteCell))
(<= whiteCell
        (cell ?x ?y)
        (true (cellHolds ?x ?y white)))
(<= blackCell
        (cell ?x ?y)
        (true (cellHolds ?x ?y black)))

(index 1) (index 2) (index 3) (index 4) (index
    ↪ 5) (index 6) (index 7) (index 8)
(succ 1 2) (succ 2 3) (succ 3 4) (succ 4 5) (
    ↪ succ 5 6) (succ 6 7) (succ 7 8)
```

Figure 7: Breakthrough in GDL (from games.ggp.org).

```
#players = white(100), black(100) // 0-100 scores
#pieces = e, w, b
#variables = // no variables
#board = rectangle(up,down,left,right,
          [b, b, b, b, b, b, b, b]
          [b, b, b, b, b, b, b, b]
          [e, e, e, e, e, e, e, e]
          [e, e, e, e, e, e, e, e]
          [e, e, e, e, e, e, e, e]
          [e, e, e, e, e, e, e, e]
          [w, w, w, w, w, w, w, w]
          [w, w, w, w, w, w, w, w])

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

#turn(me; myPawn; opp; oppPawn; forward) =
    ->me anySquare {myPawn}
    [e] forward ({e} + (left+right) {e,oppPawn})
    ->>
    [$ me=100] [$ opp=0]
    (  {? forward} [myPawn]
     + {! forward} ->> {})

#rules = (
      turn(white; w; black; b; up)
      turn(black; b; white; w; down)
    )*
```

Figure 8: Breakthrough in RBG (from RBG GitHub).

- The players must be defined before the pieces, as pieces can be assigned to players.
- The pieces must be defined before the rules, as rules are assigned to pieces.
- The players and the pieces must be defined before the postconditions, as postconditions are assigned to player/piece pairs.
- The board's size must be defined before the board itself, as exactly one piece must be placed on each square.
- The pieces and their rules must be defined before the board, as pieces are placed on the board and a player's pieces should be movable.
- Macros must be defined before they are called.

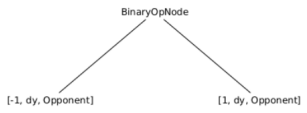
4.2.4 Generating Legal Moves

As mentioned in Section 4.2.2, a state machine is generated for each rule, which is then used to generate moves for that piece. The moves are generated by generating all words (sequences of letters) accepted by the state machine. Even though state machines can accept words of arbitrary length, most board games do not allow moves of arbitrary length. However, not all generated moves are guaranteed to be legal moves. For example, in chess, you cannot move a pinned piece, as doing so would put your king in check. To generate only legal moves we have to check the postconditions for the current player and prune the moves that do not satisfy the conditions.

```
(define "ReachedTarget" (is In (last To) (sites Mover)) )

(game "Breakthrough"
    (players {(player N) (player S)})
    (equipment {
        (board (square 8))
        (piece "Pawn" Each
            (or {
                "StepForwardToEmpty"
                (move
                    Step
                    (directions {FR FL})
                    (to if:(or
                            (is Empty (to))
                            (is Enemy (who at:(to)))
                        )
                        (apply (remove (to)))
                    )
                )
            })
        )
        (regions P1 (sites Top))
        (regions P2 (sites Bottom))
    })
    (rules
        (start {
            (place "Pawn1" (expand (sites Bottom)))
            (place "Pawn2" (expand (sites Top)))
        })

        (play (forEach Piece))

        (end (if "ReachedTarget" (result Mover Win)))
    )
)
```

Figure 9: Breakthrough in Ludii (from ludii.games).

```
macro capture(dy) = [-1, dy, Opponent] | [1, dy, Opponent];
```




```
rule wPawn = [0, 1, Empty] | [-1, 1, Empty] | [1, 1, Empty] | capture(1);
```

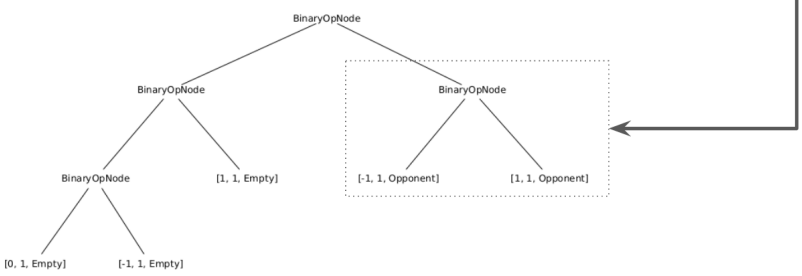


Figure 10: Resolving a macro call in a rule.

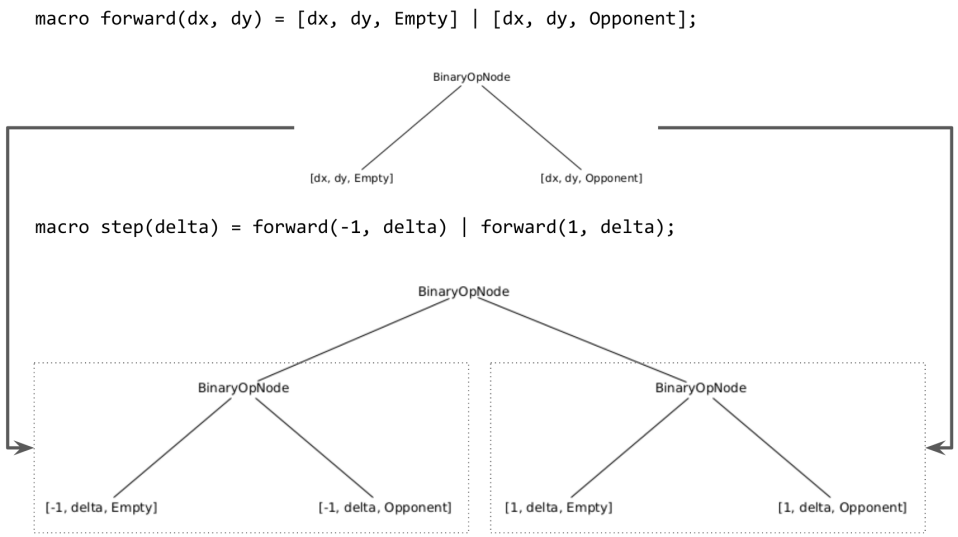


Figure 11: Resolving a macro call in a macro.

4.3 Optimizations

We decided to focus on code readability and simplicity over performance for the first iteration of our Abstract Boardgames implementation, and later focus our optimization efforts where they mattered the most. In the first iteration of our language. the whole game state was copied and stored to be able to undo moves and search the game tree. This proved to be expensive. Depicted in Figure 15 is a part of the call graph from a profile done using Callgrind during a flat Monte-Carlo experiment (discussed in Section 4.4). The call graph shows that most of the execution was spent on generating moves, but significant time was spent on memory allocation and deallocation for the game state copies.

Therefore we decided to implement "anti-effects" that store only the changes made to the game state by a move effect. This proved to be effective. Depicted in Figure 16 is a part of the call graph from a profile done under the same circumstances as the one shown in Figure 15. As an example, undoing a move only takes 19.56% of the execution with anti-effects, compared to 57.59% before, and memory allocation constitutes much less of the execution time. Now, executing a (partial) move takes a considerable portion of the execution time.

Therefore, we wanted to optimize how we generate moves next. When generating moves, we use every piece's state machine to find moves that satisfy all postconditions. To do this we execute (and undo) partial moves until we reach a terminal state in the state machine. Then we check all the postconditions to see if the move is legal. As we can see in Figure 16, executing a (partial) move takes considerable time. Now consider a state machine for a piece that has no legal moves in the current state. The piece has no legal moves either because a predicate is evaluated as false and the state machine never reached a terminal state, or none of the moves satisfied the postconditions. In the case in which a predicate is evaluated as false, we know that the predicate will not be evaluated as true unless the squares it references change. Our idea is that if we keep track of which predicates are evaluated as false and the squares they reference, we can avoid executing a piece's state machine, or at least a part of it, and save time. This would require a new parser to analyze the predicates written in C++, but unfortunately, we did not have enough time to implement this idea.

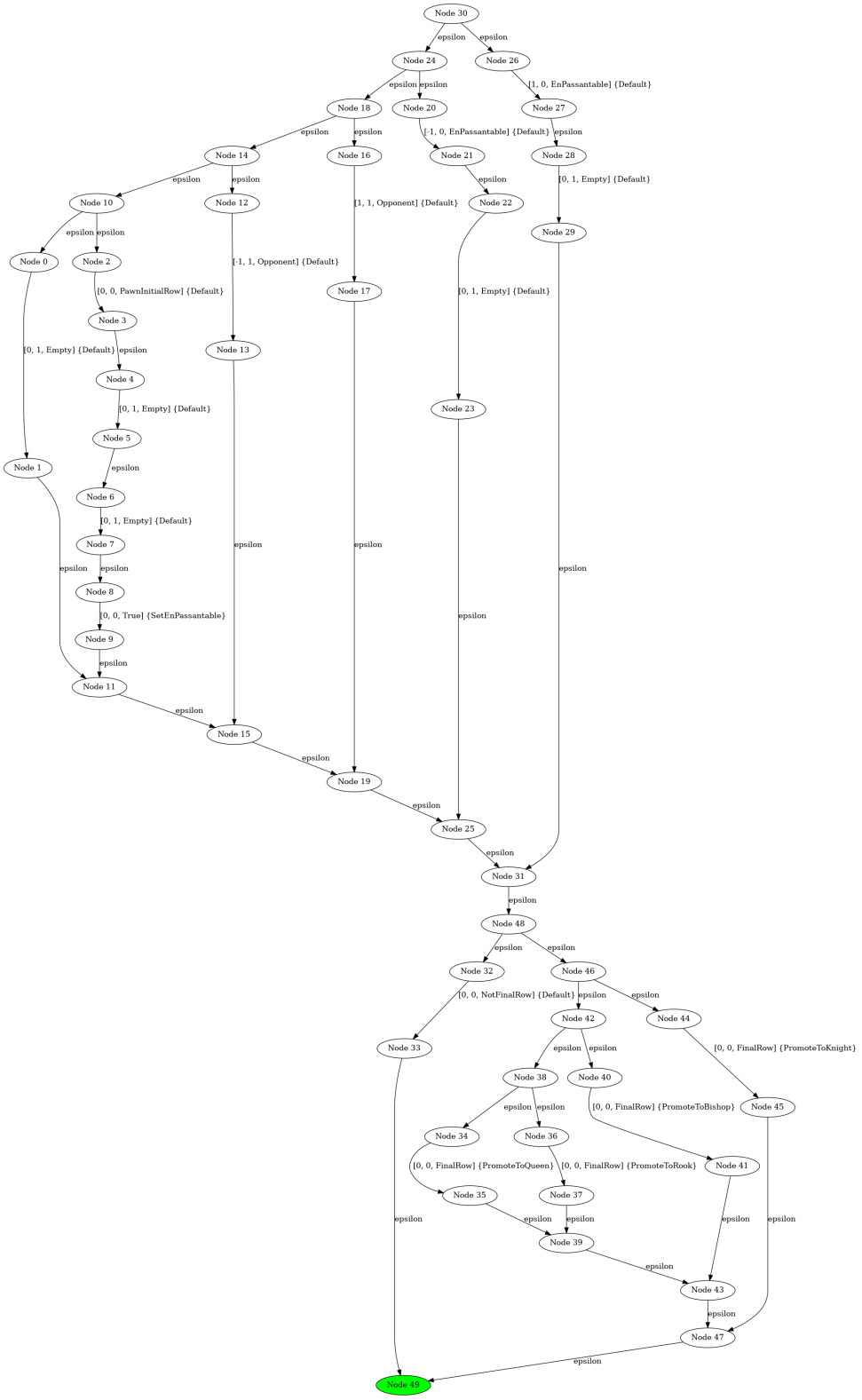


Figure 12: NFA to generate moves for a white pawn in chess.

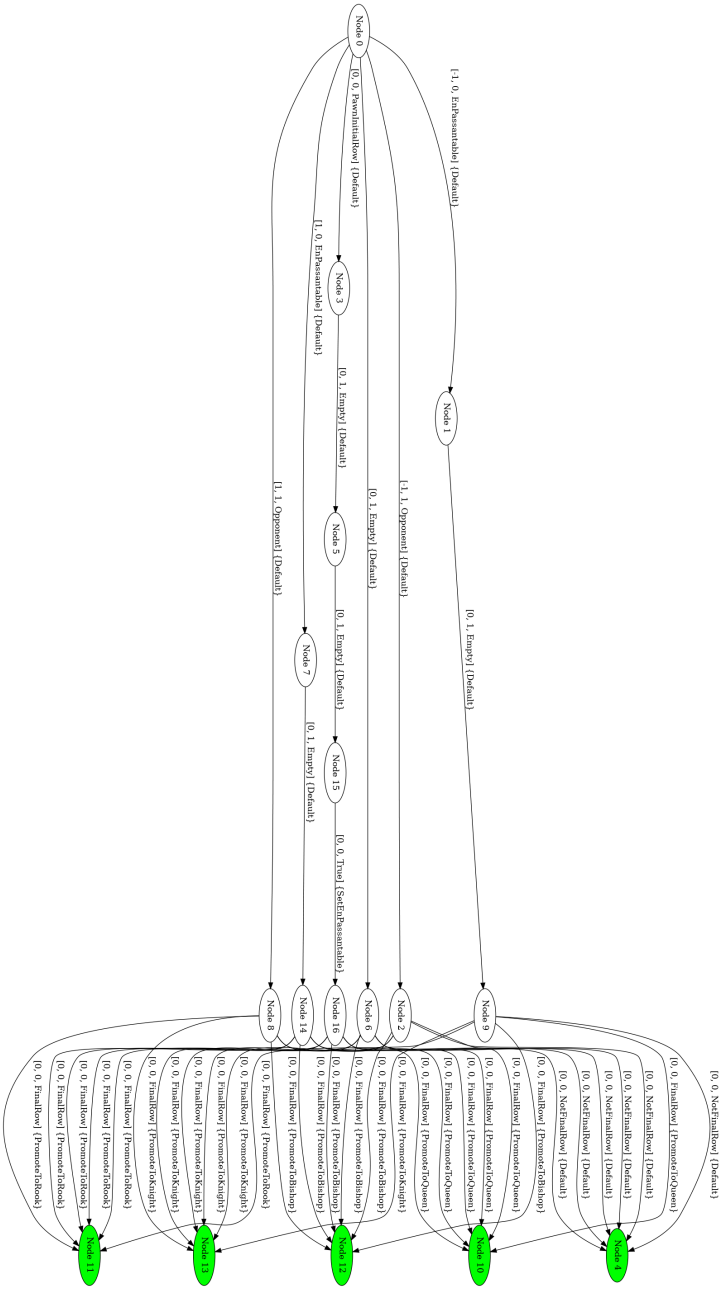


Figure 13: DFA to generate moves for a white pawn in chess.

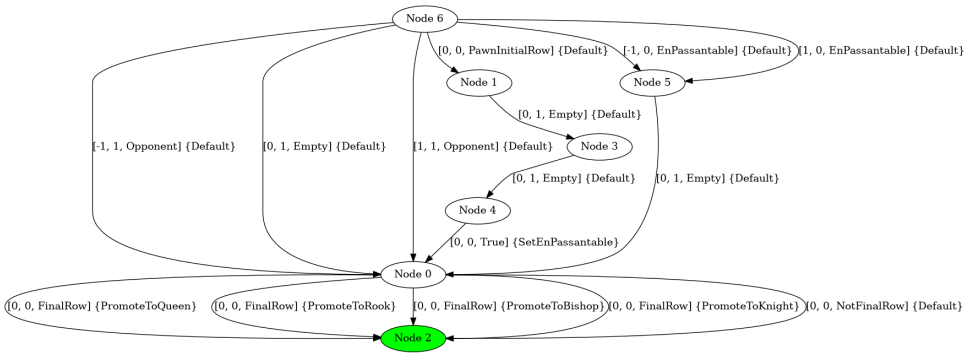


Figure 14: Minimized DFA to generate moves for a white pawn in chess.

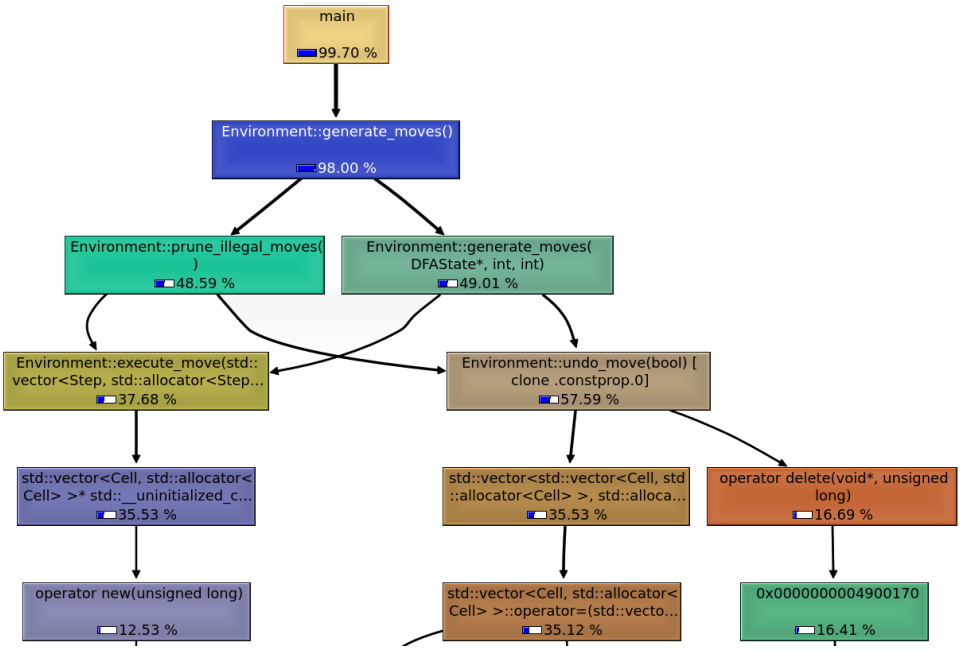


Figure 15: Part of a call graph profile of the first language implementation.

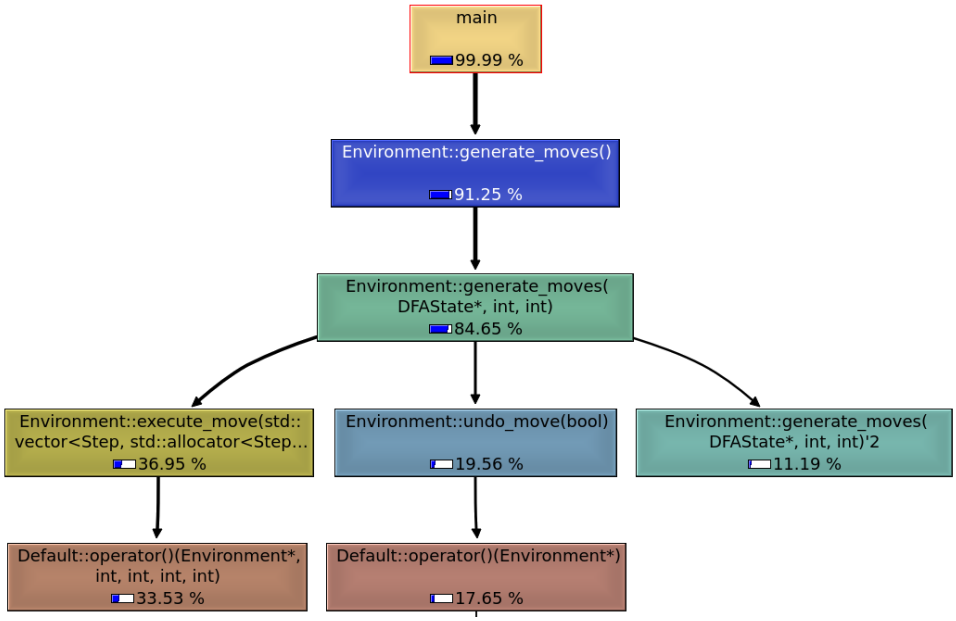


Figure 16: Part of a call graph profile after implementing anti-effects.

4.4 Experiments

In this section, we test the reasoning efficiency of our implementation of Abstract Boardgames and compare it to RBG and Ludii. We use two common efficiency metrics, *perft*, and *flat Monte-Carlo*, as is done in [8].

- **Perft** consists of computing the whole game tree to a fixed depth, starting from the initial state. Results are presented as the average number of game states computed per second.
- **Flat Monte-Carlo** consists of random rollouts from the initial state to a leaf node. In every visited state, all legal moves are computed, then one is chosen uniformly at random and played. Results are presented as the average number of states computed per second.

Flat Monte-Carlo is dominated by computing legal moves, whereas perft has a balance between computing legal moves, and making and undoing the moves. Therefore, games with large branching factors are expected to get fewer state counts per second in Flat Monte-Carlo experiments than in perft experiments.

All the experiments were done on a single core of Intel(R) Core(TM) i7-7700HQ CPU @ 2.80GHz on a laptop with 16GB of Random Access Memory (RAM). The results for perft are presented in Table 1, and the results for flat Monte-Carlo are presented in Table 2.

Table 1: Efficiency comparison between ABG, Ludii, and RBG using the perft metric, measured in states/second (higher is better).

Game	Abstract Boardgames	Ludii	RGB Interpreter	RBG Compiler
8x8 Breakthrough ($d = 6$)	1,461,606	598,262	2,671,020	36,078,575
Connect 4 ($d = 9$)	952,874	451,721	2,142,758	32,854,103
Chess ($d = 5$)	295,980	21,639	232,467	4,844,520
Tic-tac-toe (complete)	1,321,985	336,533	1,697,364	27,497,300

Table 2: Efficiency comparison between ABG, Ludii, and RBG using the flat Monte-Carlo metric, measured in states/second (higher is better).

Game	Abstract Boardgames	Ludii	RBG Interpreter	RBG Compiler
8x8 Breakthrough	107,204	191,251	302,177	2,026,175
Connect 4	207,843	514,464	1,039,628	12,686,323
Chess	5,078	29,393	17,049	239,765
Tic-tac-toe	444,683	1,382,617	1,203,842	13,542,394

Before we discuss the results we note that the Regular Boardgames project was first published over three years ago (November 2018), and has been heavily optimized through the years. The Ludii system was also first published around three years ago (May 2019) and has also been optimized. The Abstract Boardgames project was done as a proof-of-concept in a little less than three months, with little time and effort placed on optimizations.

On the perft metric, our ABG reasoners are at least two times faster than Ludii reasoners on all tested games. The greatest efficiency difference is in chess, where the ABG reasoner is over 13 times faster than the Ludii reasoner. On the flat Monte-Carlo metric, Ludii reasoners are $2-6$ times faster than ABG reasoners. This indicates that ABG can make and undo a move more efficiently than Ludii, but Ludii is more efficient at generating legal moves.

RBG provides an interpreter and a compiler to do reasoning, with the latter being significantly more efficient, and currently the fastest GGP reasoner[2].

On the perft metric, our ABG chess reasoner is 27% faster than the RBG Interpreter reasoner. In the other games, the RBG Interpreter reasoners are up to 2.25 times faster than the ABG reasoners. On the flat Monte-Carlo metric, the RBG Interpreter reasoners outperform the ABG reasoners on all tested games, with the former being $2.7-5$ times faster.

The RBG Compiler reasoners are significantly faster than the ABG reasoners on both metrics, as expected. The RBG Compiler reasoners are up to 35 times faster on the perft metric and up to 61 times faster on the flat Monte-Carlo metric.

In conclusion, we have shown that ABG reasoners achieve comparable performance to both Ludii and RBG Interpreter reasoners, with ABG reasoners outperforming the other reasoners in some games, and being the fastest in chess according to the perft metric. This suggests that our ABG implementation is already competitive with the current state of the art, with further optimization left as future work.

4.5 Towards a Self-Contained Language

While it is convenient to write predicates, move effects, and terminal conditions in C++ and reference them in ABG descriptions, that approach lacks formalism. In this section, we describe our attempt at designing a self-contained language, that is, a language that fully describes a game. To do this, we need to be able to define variables, predicates, move effects, and victory conditions in our language, that can read from and write to the current game state. To do this we extend the language's variables, and introduce *integer expressions* and *functions*. An EBNF-style grammar for the self-contained language is given in Appendix D.

Variables can now be declared in the language using a special `variables` keyword. There are also some predefined variables, such as `Board`, which is used to read from and write to the current game board, and special variables for each defined piece and player, which have attributes to access the

[2]As ABG game descriptions are interpreted it is not entirely fair to compare them to the RBG Compiler, but it is done for completeness.

piece's rule and player's score, respectively. The predefined `Board` variable returns a piece when indexed (`Board[0][0]` for example), and when a piece's rule is accessed (`wPawn.regex` for example) it is returned as a regular expression. All other variables are integers, with 0 interpreted as `False` and all other numbers interpreted as `True` when applicable. All variables can be accessed in integer expressions, predicates, move effects, and victory conditions, although predicates cannot modify them.
Integer expressions are our main tool to work with variables in the self-contained language. An integer expression always returns an integer, hence the name. An integer expression is used to compare and assign values to both the predefined and user-defined variables. An integer expression supports arithmetic operators (addition, subtraction, real and integer division, and modulo), boolean operators (`and`, `or`, and `not`), and comparison operators (equality and inequality). Within the context of boolean operators, integers are treated as boolean values.
Functions can be defined to encapsulate parameterized evaluations (integer expressions or regular expressions), and always return an integer. A function that evaluates an integer expression will return the expression's value. A function that evaluates a regular expression can only be called from a piece's rule, as the expression will be evaluated from the piece's current location on the board. The function returns whether the expression was matched or not. Within a function, two special variables are available, `Empty` and `Opponent`. They can be used to check if a square is empty or contains an opponent piece respectively.
Predicates are given as either an integer expression or a function call, whose result is interpreted as a boolean value. If the function call evaluates a regular expression, then the expression is evaluated from the piece's context, that is, its current location on the game board.
Move effects consist of a list of variable assignments. This allows us to move, remove, and add pieces to the board, set players' scores using the predefined variables, and update user-defined variables.
Victory conditions are defined for each player as an integer expression. If a player's victory condition is satisfied in some state, that player is declared the winner. This allows for multiple winners in a single state (or a draw in two-player games). If the current player has no legal moves, and no victory conditions are satisfied, the game ends in a draw.

Unfortunately, we did not have time to fully implement this idea. The current self-contained language implementation can be used to describe simple games like Breakthrough, but not more complex games like chess. The problem is that regular expression sentences can only be run in the context of some piece. To properly describe a *checkmate* terminal condition in chess, for example, a regular expression sentence would have to be run without a context. This is left as future work.

5 The Game Playing Agent

To demonstrate learning from Abstract Boardgames game descriptions we created a simple AlphaZero-style game playing agent. Like AlphaZero, our agent learns to play the game entirely from self-play, uses a neural network to evaluate game states, and uses MCTS instead of minimax-based search [6]. Our agent's neural network does not have a policy head like AlphaZero's, but only a value head. We note that the MCTS used by our agent is not identical to the description in Section 3.2.4. Instead of performing a simulation from a child of a newly expanded node, the expanded node is evaluated using the agent's neural network, and its value is propagated up the search tree. The AlphaZero paper notes that:

> *AlphaZero* evaluates positions using non-linear function approximation based on a deep neural network, rather than the linear function approximation used in typical chess programs. This provides a much more powerful representation, but may also introduce spurious approximation errors. MCTS averages over these approximation errors, which therefore

tend to cancel out when evaluating a large subtree. In contrast, alpha-beta search computes explicit minimax, which propagates the biggest approximation errors to the root of the subtree. [6]

During Breakthrough experiments, we observed that in some cases when both players were one move from victory, our minimax agent did not choose the winning move since it valued a losing move higher. In Section 5.5 we pit our minimax and MCTS agents (before and after training) against random agents.

The agent uses a C++ API implemented by ABG to interact with a game. The API is described in Appendix B.

5.1 Overview

For simplicity, our agent can only play two-player games. The rest of this chapter is structured as follows: first, we discuss what knowledge our agent has access to. Then we discuss the agent's neural network architecture, and how the training data is generated and used to train the network. Finally, we evaluate the agents' performance. Due to how the neural network is set up, we require that each game's terminal score is in the range $[-1, 1]$.

5.2 Domain Knowledge

Even though GGP agents cannot rely on domain-specific features we must provide them with a general representation of the game state so they can evaluate how good a move is. An agent can:

- query the game environment to get a general representation of the current state,
- query the game environment to get all legal moves from the current state,
- give a legal move to the game environment to execute,
- search for a good move by repeatedly getting the current state and legal moves, executing a move, and undoing it. The agent must leave the environment in the same state as before the search.

Similar to AlphaZero, the current game state is represented using a set of planes, and the agent's neural network is set up to match the shape of the state representation. A state is represented using K planes, each the same size as the game board, $N \times M$. There is one binary feature plane for each defined game piece, indicating its presence/absence, and there is one plane indicating the current player's color. Additional planes can be added depending on the game, for example, to indicate the state of special rules (legality of castling, capturing en passant, etc), as described in [6]. As an example, a 3x3 Breakthrough state is shown in Figure 17 and its representation in Table 3. The planes (from top-to-bottom) indicate the presence of black's pieces, the presence of the empty piece, the presence of white's pieces, and that it is white's turn.
The agent cannot query the game environment to ask who won in a terminal state, it must use its neural network to evaluate the states. The agent is not provided with any domain knowledge beyond the points listed above.

5.3 Architecture

In this section, we describe the agent's neural network used to evaluate game states. The network is a smaller version of the networks used by AlphaGo Zero [28] and AlphaZero [6]. The agent's neural

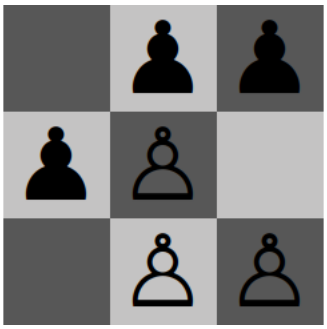


Figure 17: Example 3x3 Breakthrough state.

$$\begin{bmatrix} 0 & 1 & 1 \\ 1 & 0 & 0 \\ 0 & 0 & 0 \end{bmatrix} \begin{bmatrix} 1 & 0 & 0 \\ 0 & 0 & 1 \\ 1 & 0 & 0 \end{bmatrix} \begin{bmatrix} 0 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 1 & 1 \end{bmatrix} \begin{bmatrix} 1 & 1 & 1 \\ 1 & 1 & 1 \\ 1 & 1 & 1 \end{bmatrix}$$

Table 3: The state's (Fig 17) representation.

network consists of a single convolutional block, followed by three residual blocks, and finally a value head.

The convolutional block applies the following transformations to its input:

1. A convolution of 64 filters of kernel size 3x3 with stride and padding 1.
2. Batch normalization
3. A rectifier non-linearity.

Each residual block applies the following transformations to its input:

1. A convolution of 64 filters of kernel size 3x3 with stride and padding 1.
2. Batch normalization.
3. A rectifier non-linearity.
4. A convolution of 64 filters of kernel size 3x3 with stride and padding 1.
5. Batch normalization.
6. A skip connection that adds the original input to the block.
7. A rectifier non-linearity.

The value head applies the following transformations to its input:

1. A convolution of 1 filter of kernel size 1x1 with stride 1 and padding 0.
2. Batch normalization.
3. A rectifier non-linearity.
4. A fully connected linear layer to a hidden layer of size 64.
5. A rectifier non-linearity.

Table 4: Evaluation of agents as games won, drawn, and lost from the agent's perspective.

Game	White	Black	Win	Draw	Loss
4x4 Breakthrough	Random	Random	30	0	20
4x4 Breakthrough	Untrained MCTS	Random	46	1	3
	Random	Untrained MCTS	1	1	48
4x4 Breakthrough	Trained MCTS	Random	50	0	0
	Random	Trained MCTS	45	0	5
4x4 Breakthrough	Untrained minimax	Random	29	0	21
	Random	Untrained minimax	21	0	29
4x4 Breakthrough	Trained minimax	Random	43	0	7
	Random	Trained minimax	40	0	10

6. A fully connected linear layer to a scalar value.

7. A tanh non-linearity outputting a scalar value in the range $[-1, 1]$.

The network's complete architecture, and each block's architecture is visualized in Figures 18, 19, 20, and 21.

5.4 Creating Training Data

Training data is generated using self-play. In each training iteration, the agent plays a series of games against itself. Each game state is labeled using the final score of the game from the current player's perspective. For example, if the *black* player wins the game, then all game states in which *black* makes the next move are labeled 1, and all game states in which *white* makes the next move are labeled -1.

Only the games played during the last N training iterations are stored in a replay buffer. Using a replay buffer the agent can repeatedly learn from past experiences [29]. The idea behind only storing the most recent games is that the agent begins by playing randomly, and gradually learns to recognize good states. As the agent learns to recognize good states, it will play better games, leading the agent to find more good states. A positive feedback loop.

5.5 Experiments

We created two agents to play 4x4 Breakthrough (initial state shown in Figure 22), one which uses a minimax-based search, and one which uses MCTS search, to find the next move. Both agents were trained for 10 iterations. During each iteration, 20 games were generated as described above, and the replay buffer was set up to store the games from the last 3 iterations.

The minimax-based agent used alpha-beta pruning, searching 4 levels of the game tree, to find the next move, and the MCTS performed 50 search rounds to find the next move. This resulted in about 35 minutes of training for the minimax-based agent, and 5 minutes for the MCTS agent. We did not train the agents to achieve world-class performance, but to demonstrate an intelligent agent using our language. The training was accelerated using a `NVIDIA GTX 1050` GPU.

To evaluate the agents' performance, we pit them against random agents and keep track of how many games they win/lose as each color. Each experiment included a 50-game tournament. The agents are evaluated before and after training. The results are presented in Table 4.

To establish a baseline we pitted two random agents against each other. 4x4 Breakthrough seems to favor the starting player, who won 60% of the games when both agents played randomly.

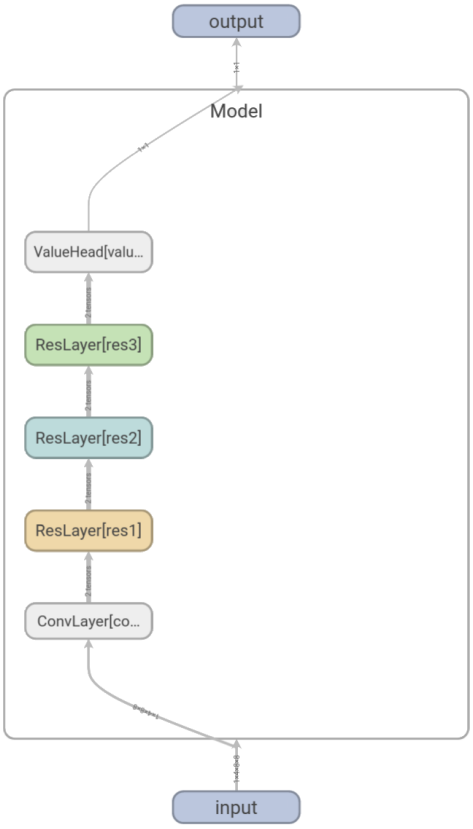


Figure 18: Complete neural network architecture.

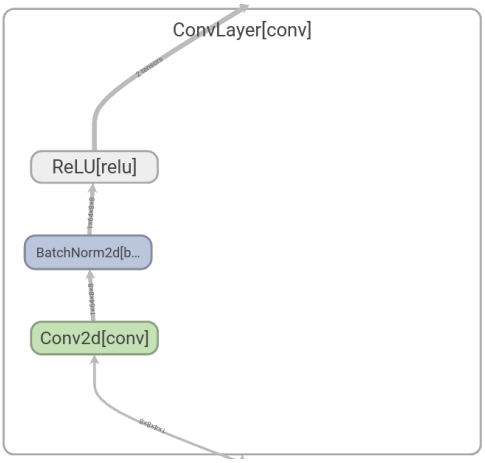


Figure 19: Convolutional block architecture.

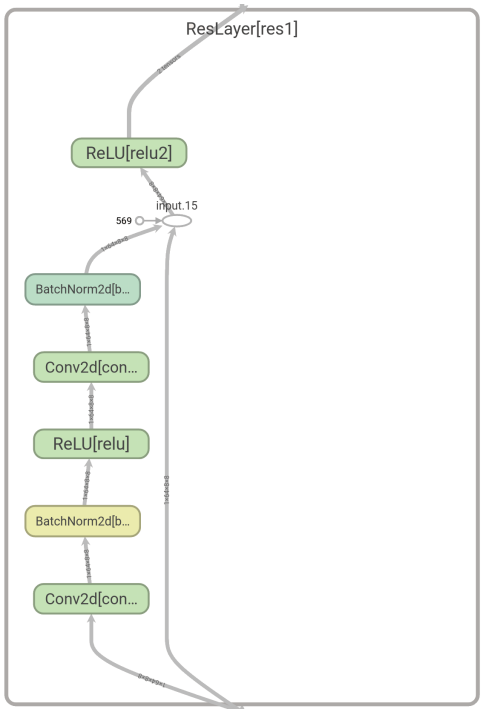


Figure 20: Residual block architecture.

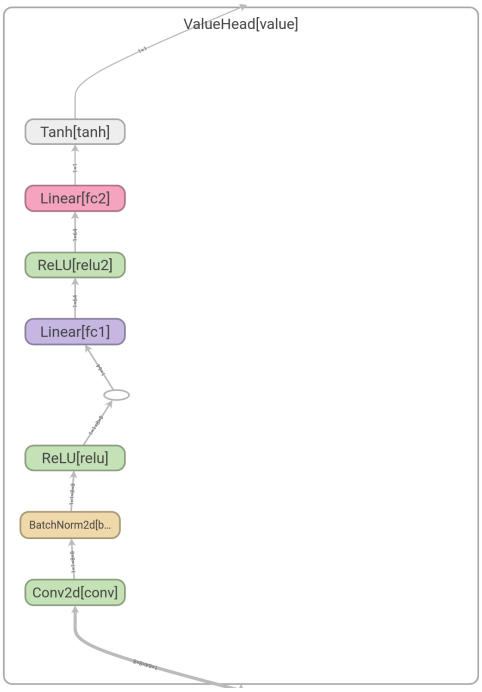


Figure 21: Value head architecture.

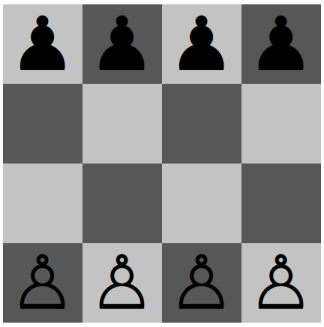


Figure 22: 4x4 Breakthrough initial state.

We observe that the AlphaZero-like MCTS agent played much better as *white* than *black* before training. We attribute this to the random initialization of the agent's neural network. After training, the agent improves as both players, winning every single game as *white*, and nearly 90% of the games as *black*, a significant improvement.

The minimax-based agent achieved similar results as a random agent before training, and improves after training, but not as much as the MCTS agent. Considering these results and the fact that the minimax-based agent requires longer training and reasoning times, and yet performs worse than the MCTS agent, we conclude both agents benefit from training but the AlphaZero-like MCTS agent is better than the minimax-based agent[3].

6 A General Game Graphical User Interface

To make playing a game defined in our language easier, we created a rudimentary web-based Graphical User Interface (GUI) using Flask, WebSockets, and Javascript. The GUI connects to our ABG language implementation, displays the current game state, and allows the current player to make a move. Using WebSockets, the GUI can query the language implementation for a list of legal moves in the current state, and when the user (human or artificial) has selected its move, it is relayed to the language implementation which executes the move and its effects. The updated game state is displayed in the GUI.

The GUI supports any two-player game written in our language as long as the relevant pieces have been defined in Javascript. As an example, chess and Tic-tac-toe are shown in our GUI in Figures 23 and 24 respectively. The grey dots show the legal moves for the currently selected piece, which has a black border around its square.

7 Conclusion and Discussion

In this report, we introduced a new game description language capable of describing a wide range of board games using what we consider to be a more user-friendly syntax than existing languages offer. We showed our language implementation to be competitive with Ludii and the RBG Interpreter in terms of reasoning efficiency, but some work still needs to be done to be competitive with the RBG Compiler. We demonstrated an intelligent agent using our language to learn to play a simple game at an above-average level from scratch, and a GUI to visually play games against humans and agents alike.

[3]Pitting the minimax-based agent against the MCTS agent does not provide interesting results as both agents play deterministically after training.

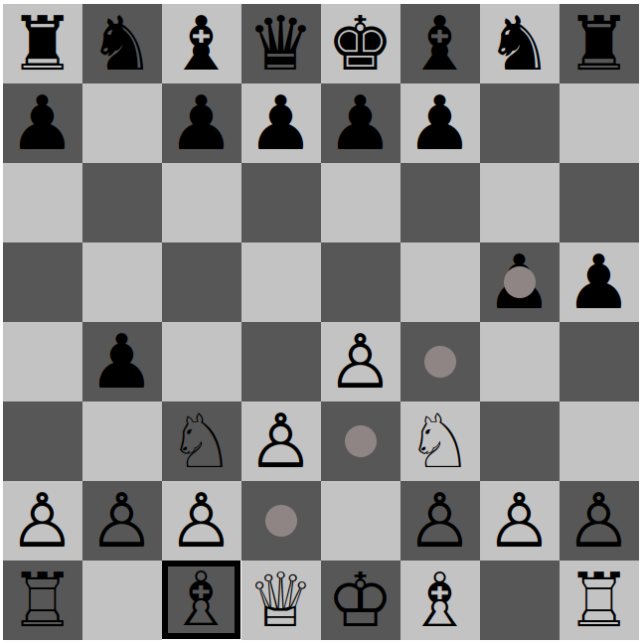


Figure 23: Chess in our GUI.

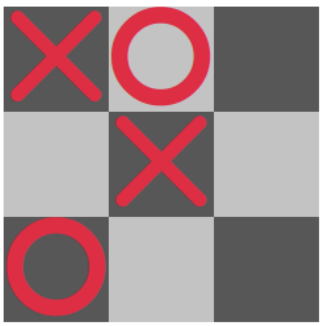


Figure 24: Tic-tac-toe in our GUI.

We consider our language to be in many ways more intuitive than the existing languages and therefore, better suited for the average AI researcher and practitioner. It just needs some more optimizations, which is outside the scope of this work, to become a truly competitive GGP language.

8 Future Work

In this section, we go over possible next steps for this project.

8.1 Optimizations

As discussed in Section 4.4, our ABG language reasoners is considerably slower than the RBG Compiler reasoners but competitive with Ludii's reasoners. In Section 4.3 we discuss what optimizations we did, and current bottlenecks in our implementation, and presented an idea to make it faster. This is left as future work.

8.2 A Self-Contained Language

In Section 4.5 we described our attempt at a self-contained language and the associated challenges. Solving the remaining challenges is left as future work. With a self-contained language, we could also look into creating ABG Compiler reasoners, similar to the RBG Compiler reasoners. Again, this is left as future work.

References

[1] Wikipedia contributors, "Board game — Wikipedia, the free encyclopedia." `https://en.wikipedia.org/w/index.php?title=Board_game&oldid=1086129523`, 2022. [Online; accessed 9-May-2022].

[2] A. M. TURING, "I.—COMPUTING MACHINERY AND INTELLIGENCE," *Mind*, vol. LIX, pp. 433–460, 10 1950.

[3] J. Schaeffer, R. Lake, P. Lu, and M. Bryant, "Chinook the world man-machine checkers champion," *AI Magazine*, vol. 17, p. 21, Mar. 1996.

[4] M. Campbell, A. J. Hoane Jr, and F.-h. Hsu, "Deep blue," *Artificial intelligence*, vol. 134, no. 1-2, pp. 57–83, 2002.

[5] D. Silver, A. Huang, C. J. Maddison, A. Guez, L. Sifre, G. Van Den Driessche, J. Schrittwieser, I. Antonoglou, V. Panneershelvam, M. Lanctot, *et al.*, "Mastering the game of go with deep neural networks and tree search," *nature*, vol. 529, no. 7587, pp. 484–489, 2016.

[6] D. Silver, T. Hubert, J. Schrittwieser, I. Antonoglou, M. Lai, A. Guez, M. Lanctot, L. Sifre, D. Kumaran, T. Graepel, *et al.*, "Mastering chess and shogi by self-play with a general reinforcement learning algorithm," *arXiv preprint arXiv:1712.01815*, 2017.

[7] N. Love, T. Hinrichs, D. Haley, E. Schkufza, and M. Genesereth, "General game playing: Game description language specification," 2008.

[8] J. Kowalski, M. Mika, J. Sutowicz, and M. Szykuła, "Regular boardgames," in *Proceedings of the AAAI Conference on Artificial Intelligence*, vol. 33, pp. 1699–1706, 2019.

[9] É. Piette, D. J. N. J. Soemers, M. Stephenson, C. F. Sironi, M. H. M. Winands, and C. Browne, "Ludii – the ludemic general game system," in *Proceedings of the 24th European Conference on Artificial Intelligence (ECAI 2020)* (G. D. Giacomo, A. Catala, B. Dilkina, M. Milano, S. Barro, A. Bugarín, and J. Lang, eds.), vol. 325 of *Frontiers in Artificial Intelligence and Applications*, pp. 411–418, IOS Press, 2020.

[10] S. Schiffel and M. Thielscher, "Representing and reasoning about the rules of general games with imperfect information," *Journal of Artificial Intelligence Research*, vol. 49, pp. 171–206, 2014.

[11] M. Thielscher, "Gdl-iii: A description language for epistemic general game playing," in *The IJCAI-16 workshop on general game playing*, p. 31, 2017.

[12] E. Piette, M. Stephenson, D. J. Soemers, and C. Browne, "An empirical evaluation of two general game systems: Ludii and rbg," in *2019 IEEE Conference on Games (CoG)*, pp. 1–4, IEEE, 2019.

[13] S. Schiffel and M. Thielscher, "Fluxplayer: A successful general game player," in *Aaai*, vol. 7, pp. 1191–1196, 2007.

[14] J. Clune, "Heuristic evaluation functions for general game playing," in *AAAI*, vol. 7, pp. 1134–1139, 2007.

[15] Y. Bjornsson and H. Finnsson, "Cadiaplayer: A simulation-based general game player," *IEEE Transactions on Computational Intelligence and AI in Games*, vol. 1, no. 1, pp. 4–15, 2009.

[16] Wikipedia contributors, "Game tree — Wikipedia, the free encyclopedia." https://en.wikipedia.org/w/index.php?title=Game_tree&oldid=1081174077, 2022. [Online; accessed 1-May-2022].

[17] Wikipedia contributors, "Minimax — Wikipedia, the free encyclopedia." https://en.wikipedia.org/w/index.php?title=Minimax&oldid=1076761456, 2022. [Online; accessed 1-May-2022].

[18] Wikipedia contributors, "Alpha–beta pruning — Wikipedia, the free encyclopedia." https://en.wikipedia.org/w/index.php?title=Alpha%E2%80%93beta_pruning&oldid=1075297799, 2022. [Online; accessed 2-May-2022].

[19] L. Kocsis and C. Szepesvári, "Bandit based monte-carlo planning," in *European conference on machine learning*, pp. 282–293, Springer, 2006.

[20] Wikipedia contributors, "Monte carlo tree search — Wikipedia, the free encyclopedia." https://en.wikipedia.org/w/index.php?title=Monte_Carlo_tree_search&oldid=1084380056, 2022. [Online; accessed 2-May-2022].

[21] Wikipedia contributors, "Breakthrough (board game) — Wikipedia, the free encyclopedia." https://en.wikipedia.org/w/index.php?title=Breakthrough_(board_game)&oldid=1067107785, 2022. [Online; accessed 12-May-2022].

[22] Wikipedia contributors, "Lexical analysis — Wikipedia, the free encyclopedia." https://en.wikipedia.org/w/index.php?title=Lexical_analysis&oldid=1077694767, 2022. [Online; accessed 2-May-2022].

[23] Wikipedia contributors, "Parsing — Wikipedia, the free encyclopedia." https://en.wikipedia.org/w/index.php?title=Parsing&oldid=1076332498, 2022. [Online; accessed 2-May-2022].

[24] Wikipedia contributors, "Ll grammar — Wikipedia, the free encyclopedia." https://en.wikipedia.org/w/index.php?title=LL_grammar&oldid=1073763378, 2022. [Online; accessed 2-May-2022].

[25] Wikipedia contributors, "Ll parser — Wikipedia, the free encyclopedia." https://en.wikipedia.org/w/index.php?title=LL_parser&oldid=1077248773, 2022. [Online; accessed 2-May-2022].

[26] Wikipedia contributors, "Nondeterministic finite automaton — Wikipedia, the free encyclopedia." https://en.wikipedia.org/w/index.php?title=Nondeterministic_finite_automaton&oldid=1085481352, 2022. [Online; accessed 2-May-2022].

[27] Wikipedia contributors, "Powerset construction — Wikipedia, the free encyclopedia." https://en.wikipedia.org/w/index.php?title=Powerset_construction&oldid=1028566254, 2021. [Online; accessed 2-May-2022].

[28] D. Silver, J. Schrittwieser, K. Simonyan, I. Antonoglou, A. Huang, A. Guez, T. Hubert, L. Baker, M. Lai, A. Bolton, *et al.*, "Mastering the game of go without human knowledge," *nature*, vol. 550, no. 7676, pp. 354–359, 2017.

[29] L.-J. Lin, "Self-improving reactive agents based on reinforcement learning, planning and teaching," *Machine learning*, vol. 8, no. 3, pp. 293–321, 1992.

Appendices

Appendix A Abstract Boardgames Grammar

game:	player_list ";" piece_list ";" (macro_def ";")* (rule ";")* (post_condition ";")* board_size ";" board ";"
player_list:	"players" "=" player ("," player)*
player:	string
piece_list:	"pieces" "=" piece ("," piece)*
piece:	string ("(" piece_owners ")")?
piece_owner:	player, ("," player)*
board_size:	"board_size" "=" int "," int
board:	"board" "=" board_piece ("," board_piece)*
board_piece:	string
rule:	"rule" piece "=" sentence
macro_def:	"macro" macro_name "(" arguments? ")" = macro_sentence
macro_call:	macro_name "(" arguments? ")"
macro_name:	string
arguments:	argument ("," argument)*
argument:	int | string
macro_sentence:	macro_word ("|" macro_word)*
macro_word:	(macro_core_word unary_op | macro_core_word | macro_call)+
macro_core_word:	"(" macro_sentence ")" | macro_letter
macro_letter:	"[" string "," string "," string "]" ("" string "")?
post_condition:	"post" player piece "=" sentence
sentence:	word ("|" word)*
word:	(core_word unary_op | core_word | macro_call)+
core_word:	"(" sentence ")" | letter
letter:	"[" int "," int "," string "]" ("" string "")?
int:	CLASS OF INTEGERS
string:	CLASS OF STRINGS WITHOUT INTEGERS
unary_op:	"*" | "?" | "+"

Table 5: Abstract Boardgames EBNF-style grammar

Appendix B C++ ABG API

The API functions relevant to searching are:

- `generate_moves()`, which returns a list of legal moves in the current game state.
- `execute_move(move)`, which executes a legal move in the current game state.
- `undo_move()`, which undoes the last executed move.
- `game_over()`, which returns `True` if the current game state is a terminal state.

The API functions relevant to a learning agent (assumes two-player games) are:

- `get_environment_representation()`, which returns a 3-dimensional representation of the current game state.
- `get_first_player()`, which returns the player who made the first move.
- `get_current_player()`, which returns the player who is to make the next move.
- `get_white_score()`, which returns the score of the *white* player in the current state.
- `reset()`, which resets the game to its initial state.

Appendix C Game Descriptions in ABG

In this appendix we include all game descriptions written in Abstract Boardgames to date and a description of their predicates, move effects, extra user-defined variables (if any), and terminal conditions. The games described demonstrate four different but important behaviors in board games. In Breakthrough, every piece begins on the board, and the pieces only move around. In chess, we have rules that do not depend solely on the current board configuration, but also on information from past states, such as castling and capturing en passant. We also have promotions, which demonstrate replacing an existing piece with a new one. Finally, Connect 4 and Tic-tac-toe demonstrate placing new pieces on the board.

We have verified experimentally that the state space defined by our descriptions is the same as is defined by RBG descriptions, by comparing the two-game trees to a fixed depth.

There are a few variables that are defined for every Abstract Boardgames game description:

- One variable for each player to keep track of the player's score in the current game state.
- One boolean variable set to `True` if and only if the current game state is a terminal state.
- One variable to keep track of the number of legal moves in the current game state. Necessary to determine stalemates.

The descriptions, and the source code for the predicates, move effects, and terminal conditions are also available on GitHub.

C.1 Breakthrough

A description of Breakthrough in Abstract Boardgames is shown in Figure 25. We need two predicates to describe Breakthrough:

- **Empty**, which is true if and only if the destination square contains the special *empty* piece.
- **Opponent**, which is true if and only if the destination square contains an opponent piece.

There is only one move effect:

- **Default**, which moves your piece from its source square to its destination square, and if applicable, captures an opponent's piece on the destination square.

There are three terminal conditions:

- **Black reaches white's home row**, and wins the game.
- **White reaches black's home row**, and wins the game.
- **The current player has no legal move**, and loses the game.

C.2 Chess

A description of chess in Abstract Boardgames is shown in Figure 26. It is the most complicated game currently implemented in Abstract Boardgames. We need a few variables to effectively describe some rules in chess:

- One variable to keep track of which pawn, if any, can be captured en passant.

```
players = white, black;

pieces = empty, wPawn(white),
                bPawn(black);

macro forward(dy) = [0, dy, Empty] | [-1, dy, Empty] | [1, dy, Empty];
macro capture(dy) = [-1, dy, Opponent] | [1, dy, Opponent];

rule bPawn = forward(-1) | capture(-1);
rule wPawn = forward( 1) | capture( 1);

board_size = 8, 8;
board = bPawn, bPawn,   bPawn,   bPawn,   bPawn, bPawn,   bPawn,   bPawn,
        bPawn, bPawn,   bPawn,   bPawn,   bPawn, bPawn,   bPawn,   bPawn,
        empty, empty,   empty,   empty,   empty, empty,   empty,   empty,
        empty, empty,   empty,   empty,   empty, empty,   empty,   empty,
        empty, empty,   empty,   empty,   empty, empty,   empty,   empty,
        empty, empty,   empty,   empty,   empty, empty,   empty,   empty,
        wPawn, wPawn,   wPawn,   wPawn,   wPawn, wPawn,   wPawn,   wPawn,
        wPawn, wPawn,   wPawn,   wPawn,   wPawn, wPawn,   wPawn,   wPawn;
```

Figure 25: Breakthrough in ABG.

- One variable for each king and rook to keep track if they have been moved. Necessary for castling.

We need a few predicates to describe chess:

- **True**, which is always true.
- **Empty**, which is true if and only if the destination square contains the special *empty* piece.
- **Opponent**, which is true if and only if the destination square contains an opponent piece.
- One predicate for each of the defined pieces to check its presence on a square. Necessary for the postcondition that makes sure a player's king is not in check after making a move.
- **PawnInitialRow**, which is true if and only if a pawn can move two squares in one move.
- **FinalRow**, which is true if and only if a piece is in the opposite final row. Necessary for promotions.
- **EnPassantable**, which is true if and only if the piece on the destination square can be captured en passant.
- **RightToCastleLeft**, which is true if and only if the current player has the right to castle left.
- **RightToCastleRight**, which is true if and only if the current player has the right to castle right.
- **NotAttacked**, which is true if and only if the destination square is not under attack. Necessary to determine if castling is legal.

We need a few move effects to describe chess:

- **Default**, which moves your piece from its source square to its destination square, and if applicable, captures an opponent's piece on the destination square.

- Effects to promote a pawn to a queen, a rook, a bishop, and a knight.
- **SetEnPassantable**, which marks a piece so it can be captured en passant.
- Effects to castle left and to castle right.
- **MarkMoved**, which marks a piece as having been moved. Necessary for castling.

There is are two terminal conditions:

- **The current player has no legal moves**. If the current player is in check, the opponent wins, else the game is a draw.
- **50 moves without moving a pawn or capturing**, resulting in a draw.

C.3 Connect 4

A description of Connect 4 in Abstract Boardgames is shown in Figure 27. We need two predicates to describe Connect 4:

- **False**, which is always false.
- **LowestUnoccupied**, which is true if and only if the destination square is the lowest empty square in its column.

There is only one move effect:

- **Default**, which places the current player's piece on the destination square.

There are two terminal conditions:

- **The current player has four pieces connected**, and wins the game.
- **The current player has no legal move**, resulting in a draw.

C.4 Tic-Tac-Toe

A description of Tic-tac-toe in Abstract Boardgames is shown in Figure 28. We need two predicates to describe Tic-tac-toe:

- **False**, which is always false.
- **True**, which is always true.

There is only one move effect:

- **Default**, which places the current player's piece on the destination square.

There are two terminal conditions:

- **The current player has three pieces connected**, and wins the game.
- **The current player has no legal move**, resulting in a draw.

```
players = white, black;

pieces = empty, bRook(black), bKnight(black), bBishop(black), bQueen(black), bKing(black), bPawn(black),
                wRook(white), wKnight(white), wBishop(white), wQueen(white), wKing(white), wPawn(white);

macro step(dx, dy) = [dx, dy, Empty] | [dx, dy, Opponent];
macro line(dx, dy) = [dx, dy, Empty]*([dx, dy, Empty] | [dx, dy, Opponent]);

macro pawnStep(dy) =
        [0, dy, Empty] | [0, 0, PawnInitialRow][0, dy, Empty][0, dy, Empty][0, 0, True]{SetEnPassantable} |
        [-1, dy, Opponent] | [1, dy, Opponent] |
        [-1, 0, EnPassantable][0, dy, Empty] | [1, 0, EnPassantable][0, dy, Empty];

macro rookMoves() = line(1, 0) | line(-1, 0) | line(0, 1) | line(0, -1);

macro bishopMoves() = line(1, 1) | line(1, -1) | line(-1, 1) | line(-1, -1);

macro knightJump(dx, dy) = [dx, dy, Empty] | [dx, dy, Opponent];

macro knightMoves() = knightJump(1,  2) | knightJump(-1,  2) | knightJump(1, -2) | knightJump(-1, -2) |
                      knightJump(2,  1) | knightJump(-2,  1) | knightJump(2,  1) | knightJump(-2,  1) |
                      knightJump(2, -1) | knightJump(-2, -1);

macro kingMoves() = step(1,  1) | step(-1,  1) |
                    step(1, -1) | step(-1, -1) |
                    step(1,  0) | step(-1,  0) |
                    step(0, -1) | step( 0,  1);

macro castleMoves() =
        [0, 0, RightToCastleRight][0, 0, NotAttacked][1, 0, Empty][0, 0, NotAttacked][1, 0, Empty]
            [0, 0, NotAttacked]{CastleRight} |
        [0, 0, RightToCastleLeft][0, 0, NotAttacked][-1, 0, Empty][0, 0, NotAttacked][-1, 0, Empty]
            [0, 0, NotAttacked][-1, 0, Empty][1, 0, True][0, 0, True]{CastleLeft};

macro promotePawn() = [0, 0, FinalRow]{PromoteToQueen} | [0, 0, FinalRow]{PromoteToRook} |
                      [0, 0, FinalRow]{PromoteToBishop} | [0, 0, FinalRow]{PromoteToKnight};

macro attackedByBPawn()   = [-1,  1, BPawn]   | [ 1,  1, BPawn];
macro attackedByBKnight() = [-1, -2, BKnight] | [ 1, -2, BKnight] | [ 2, -1, BKnight] | [ 2,  1, BKnight] |
                            [ 1,  2, BKnight] | [-1,  2, BKnight] | [-2,  1, BKnight] | [-2, -1, BKnight];
macro attackedByBBishop() = [ 1,  1, Empty]*[ 1,  1, BBishop] | [ 1, -1, Empty]*[ 1, -1, BBishop] |
                            [-1, -1, Empty]*[-1, -1, BBishop] | [-1,  1, Empty]*[-1,  1, BBishop];
macro attackedByBRook()   = [ 1,  0, Empty]*[ 1,  0, BRook] | [ 0, -1, Empty]*[ 0, -1, BRook] |
                            [-1,  0, Empty]*[-1,  0, BRook] | [ 0,  1, Empty]*[ 0,  1, BRook];
macro attackedByBQueen()  =
        [ 1,  1, Empty]*[ 1,  1, BQueen] | [ 1, -1, Empty]*[ 1, -1, BQueen] | [-1, -1, Empty]*[-1, -1, BQueen] |
        [-1,  1, Empty]*[-1,  1, BQueen] | [ 1,  0, Empty]*[ 1,  0, BQueen] | [ 0, -1, Empty]*[ 0, -1, BQueen] |
        [-1,  0, Empty]*[-1,  0, BQueen] | [ 0,  1, Empty]*[ 0,  1, BQueen];
macro attackedByBKing()   = [ 1,  1, BKing] | [ 1, -1, BKing] | [-1, -1, BKing] | [-1,  1, BKing] |
                            [ 1,  0, BKing] | [ 0, -1, BKing] | [-1,  0, BKing] | [ 0,  1, BKing];

macro attackedByWPawn()   = [-1, -1, WPawn] | [1, -1, WPawn];
macro attackedByWKnight() = [-1, -2, WKnight] | [ 1, -2, WKnight] | [ 2, -1, WKnight] | [ 2,  1, WKnight] |
                            [ 1,  2, WKnight] | [-1,  2, WKnight] | [-2,  1, WKnight] | [-2, -1, WKnight];
macro attackedByWBishop() = [ 1,  1, Empty]*[ 1,  1, WBishop] | [ 1, -1, Empty]*[ 1, -1, WBishop] |
                            [-1, -1, Empty]*[-1, -1, WBishop] | [-1,  1, Empty]*[-1,  1, WBishop];
macro attackedByWRook()   = [ 1,  0, Empty]*[ 1,  0, WRook] | [ 0, -1, Empty]*[ 0, -1, WRook] |
                            [-1,  0, Empty]*[-1,  0, WRook] | [ 0,  1, Empty]*[ 0,  1, WRook];
macro attackedByWQueen()  =
        [ 1,  1, Empty]*[ 1,  1, WQueen] | [ 1, -1, Empty]*[ 1, -1, WQueen] | [-1, -1, Empty]*[-1, -1, WQueen] |
        [-1,  1, Empty]*[-1,  1, WQueen] | [ 1,  0, Empty]*[ 1,  0, WQueen] | [ 0, -1, Empty]*[ 0, -1, WQueen] |
        [-1,  0, Empty]*[-1,  0, WQueen] | [ 0,  1, Empty]*[ 0,  1, WQueen];
macro attackedByWKing()   = [ 1,  1, WKing] | [ 1, -1, WKing] | [-1, -1, WKing] | [-1,  1, WKing] |
                            [ 1,  0, WKing] | [ 0, -1, WKing] | [-1,  0, WKing] | [ 0,  1, WKing];

rule bRook   = rookMoves()[0, 0, True]{MarkMoved};
rule bKnight = knightMoves();
rule bBishop = bishopMoves();
rule bQueen  = rookMoves() | bishopMoves();
rule bKing   = (kingMoves() | castleMoves())[0, 0, True]{MarkMoved};
rule bPawn   = pawnStep(-1)([0, 0, NotFinalRow] | promotePawn());

rule wRook   = rookMoves()[0, 0, True]{MarkMoved};
rule wKnight = knightMoves();
rule wBishop = bishopMoves();
rule wQueen  = rookMoves() | bishopMoves();
rule wKing   = (kingMoves() | castleMoves())[0, 0, True]{MarkMoved};
rule wPawn   = pawnStep(1)([0, 0, NotFinalRow] | promotePawn());

post black bKing = attackedByWPawn()  | attackedByWKnight() | attackedByWBishop() | attackedByWRook() |
                   attackedByWQueen() | attackedByWKing();

post white wKing = attackedByBPawn()  | attackedByBKnight() | attackedByBBishop() | attackedByBRook() |
                   attackedByBQueen() | attackedByBKing();

board_size = 8, 8;
board = bRook, bKnight, bBishop, bQueen, bKing, bBishop, bKnight, bRook,
        bPawn, bPawn,   bPawn,   bPawn,  bPawn, bPawn,   bPawn,   bPawn,
        empty, empty,   empty,   empty,  empty, empty,   empty,   empty,
        empty, empty,   empty,   empty,  empty, empty,   empty,   empty,
        empty, empty,   empty,   empty,  empty, empty,   empty,   empty,
        empty, empty,   empty,   empty,  empty, empty,   empty,   empty,
        wPawn, wPawn,   wPawn,   wPawn,  wPawn, wPawn,   wPawn,   wPawn,
        wRook, wKnight, wBishop, wQueen, wKing, wBishop, wKnight, wRook;
```

Figure 26: Chess in ABG (more readable on GitHub).

```
players = white, black;

pieces = empty(black, white), bPawn(black), wPawn(white);

rule empty = [0, 0, LowestUnoccupied];
rule bPawn = [0, 0, False];
rule wPawn = [0, 0, False];

board_size = 7, 6;
board = empty, empty, empty, empty, empty, empty, empty,
        empty, empty, empty, empty, empty, empty, empty,
        empty, empty, empty, empty, empty, empty, empty,
        empty, empty, empty, empty, empty, empty, empty,
        empty, empty, empty, empty, empty, empty, empty,
        empty, empty, empty, empty, empty, empty, empty;
```

Figure 27: Connect 4 in ABG.

```
players = white, black;

pieces = empty(black, white), o(black), x(white);

rule empty = [0, 0, True];
rule x     = [0, 0, False];
rule o     = [0, 0, False];

board_size = 3, 3;
board = empty, empty, empty,
        empty, empty, empty,
        empty, empty, empty;
```

Figure 28: Tic-Tac-Toe in ABG.

Appendix D Self-contained Abstract Boardgames Grammar

game:	player_list ";" piece_list ";" variable* ";" (macro_def ";")* (function ";")* (effect ";")* (victory_condition ";")* (rule ";")* (post_condition ";")* board_size ";" board ";"
player_list:	"players" "=" player ("," player)*
player:	string
piece_list:	"pieces" "=" piece ("," piece)*
piece:	string ("(" piece_owners ")")?
piece_owner:	player, ("," player)*
board_size:	"board_size" "=" int "," int
board:	"board" "=" board_piece ("," board_piece)*
board_piece:	string
variable:	"variable" variable_name "=" int_expression ";"
variable_name:	string
function:	"function" function_name "(" arguments? ")" function_body ";"
function_name:	string
function_body:	"{" "return" int_expression "}" | "=" sentence
function_call:	function_name "(" function_arguments? ")"
function_arguments:	int_expression ("," int_expression)*
effect:	"effect" effect_name "(" arguments? ")" "{" effect_body "}" ";"
effect_name:	string
effect_body:	assignment*
assignment:	variable_name ("[" int_expression "]")* "=" assignment_body";"
assignment_body:	int_expression | variable_name ("[" int_expression "]")*
effect_call:	effect_name "(" function_arguments? ")"
victory_condition:	"victory" player "{" "return" int_expression "}"
rule:	"rule" piece "=" sentence
macro_def:	"macro" macro_name "(" arguments? ")" = sentence
macro_call:	macro_name "(" arguments? ")"

macro_name:	string
post_condition:	"post" player piece "=" sentence
sentence:	word ("|" word)*
word:	(core_word unary_op | core_word | macro_call)+
core_word:	"(" sentence ")" | letter | function_call | variable
letter:	"[" int_expression "," int_expression "," function_call "]" ("{" effect_call "}")?
int_expression:	unary_expression (binary_exp_op unary_expression)*
unary_expression:	unary_exp_op unary_expression | core_expression
core_expression:	"(" int_expression ")" | int | function_call | variable_name (("." variable_name)? | ("[" int_expression "]")*)
binary_exp_op:	"*" | "+" | "-" | "//" | "/" | "%" | "and" | "or" | "!=" | "==" | "&&" | "||"
unary_expr_op:	"-" | "not" | "!"
arguments:	argument ("," argument)*
argument:	int | string
int:	CLASS OF INTEGERS
string:	CLASS OF STRINGS WITHOUT INTEGERS
unary_op:	"*" | "?" | "+"

Table 6: Self-contained Abstract Boardgames EBNF-style grammar