Learning Player Strategies Using Weak Constraints

Author
ELLIOT GREENWOOD

Supervisor
DR. KRYSIA BRODA

Co-Supervisor
MARK LAW

Second Marker
PROF. ALESSANDRA RUSSO

IMPERIAL COLLEGE LONDON
DEPARTMENT OF COMPUTING

18th June 2018

ABSTRACT

Computer programs are now able to easily beat human world champions at complex games like *Chess* and *Go*, but the reasoning behind *how* these programs choose their moves remains relatively obscure. This is due to popular methods using neural networks, which are difficult to interpret without heavy analysis.

In this work we explore an alternative method of learning strategies from gameplay using Inductive Logic Programming (ILP), a logical learning framework that has been extended to the Answer Set Programming (ASP) paradigm. Specifically, we use *weak constraints* from ASP to learn preferences between board states based on human-driven examples. Learning weak constraints is a novel technique that has recently been introduced into ILASP, a learning system created at Imperial College London.

We discuss and show through experimentation ILASP's suitability to learning strategies through weak constraints. We provide a methodology for describing, abstractly, game mechanics using the Game Description Language, learning the rules of the game, strategies to use, and employing the learnt strategies with the aid of a planner. In game theory, search trees are used to employ minimax, a way of expressing optimal play by reasoning about the value of moves in the future. We take this notion and extend existing ILP frameworks to rank moves with respect to game trees.

ACKNOWLEDGEMENTS

I would like to thank my supervisor Dr. Krysia Broda for her guidance, advice and enthusiasm throughout the project. I wish to express my gratitude to Mark Law for the time and help he has dedicated towards this project and allowing me to use ILASP, he has also given me countless ideas during the last year.

I also wish to thank Lottie, for her incredible support and patience over the past few years.

Thank you to Jack and Dan, for all your help and making me laugh. For playing hours of games against me, I would like to say thank you to James, Michael, Florian, Jonathan.

Finally, thank you to my parents, my sister, and my friends for their encouragement during my time at Imperial.

CONTENTS

LIST OF FIGURES

LIST OF TABLES

LIST OF LOGIC PROGRAMS

LIST OF ALGORITHMS

LIST OF DEFINITIONS

ACRONYMS

AI Artificial Intelligence

ASP Answer Set Programming

CDOE Context Dependent Ordering Example

CDPI Context Dependent Partial Interpretation

GDL Game Description Language

GGP General Game Playing

ILASP Inductive Learning of Answer Set Programs

ILP Inductive Logic Programming

KIF Knowledge Interchange Format

LAS Learning from Answer Sets

LOAS Learning from Ordered Answer Sets

NAF Negation as Failure

PGN Portable Game Notation

Part I

PRELIMINARIES

1 INTRODUCTION

Humans have been playing and mastering games for thousands of years. The earliest games date back to the ancient Egyptian and Mesopotamian era, with games such as *Senet*[1] and the *Royal Game of Ur*[2]. Nowadays, people all over the world compete in fierce championships to prove themselves to be the best *Chess* and *Go* players, but in recent years computers have begun to surpass human level skill in many games (Mnih et al., 2013; Silver et al., 2017b). The rate at which these programs are learning is incredible, with world champions describing the games they play as alien (Peter Heine Nielsen, 2017). However, the moves that the programs make appear counter-intuitive or simply inexplicable even by masters in the field, such as Michael Redmond[3].

In this project we explore learning game strategies using a Inductive Logic Programming (ILP) system, ILASP (Law, Russo and Broda, 2014), built on top of a logical paradigm known as Answer Set Programming (ASP). We utilise *weak constraints* as the method of comparing board states and explore what strategies, if any, can be expressed using them. The advantages of using a logical framework is that they are explainable. Strategies that can be learnt in an ILP system can be conveyed in English.

1.1 MOTIVATION

Inductive Learning of Answer Set Programs (ILASP) has recently been extended to learn preferences through weak constraints (Law, Russo and Broda, 2015a). By specifying an ordering over particular concrete examples it learns generic rules that can be used to score future examples. Whilst it has been used in some small cases to learn very guided hypotheses (Law, Russo and Broda, 2015b), in this project we aim to use ILASP to learn about more open ended, practical problems.

Board games have long been used in the field of machine learning as a way of demonstrating the power of a particular system, e. g. Reinforcement Learning in *AlphaZero* to learn the game *Go*. Games are a convenient way of demonstrating machine learning for several reasons:

- they are easy to follow,
- it is easy to determine the success or failure of the agent,
- the environment is self-contained and therefore fairly simple.

Learning strategies in board games is a domain specific view of the more general problem of *preference learning*. When a player chooses a move to play they are expressing their preference for that move over other possible options. Preference learning is a problem that has been attempted in many fields for other purposes (e. g. PageRank Algorithms, Liu (2007)). Preference learning in ILASP tries to find a minimal hypothesis that *exactly*

---

[1] `https://www.boardgamegeek.com/boardgame/2399/senet`

[2] `http://www.britishmuseum.org/research/collection_online/collection_object_details.aspx?objectId=8817&partId=1`

[3] When talking about move 37 of game 2, *AlphaGo* vs. Lee Sedol

describes the non-noisy examples, the hypotheses can contain variables, allowing them to generalise over other examples. We explore the effectiveness and suitability of using ILASP, developed by Law, Russo and Broda (2014) at Imperial College London, as a solution to this problem.

In this project we primarily use the strategy game *Onitama* to test and explore ILASP. *Onitama* was chosen as it has not been solved but is not as complex as *Chess*. The game also is a little different to traditional abstract strategy games. Specifically, the movement of the pieces changes every turn (in a predictable manner) through the use of a set of cards and so having a strategy that prefers a particular piece is not viable (cf. the Queen in *Chess*), instead ILASP will need to learn higher level movement features. Additionally, *Onitama* has a random set up and so there is no similar notion of standard opening theory that exists in *Chess*.

1.2 OBJECTIVES

The objectives for this project are:

- formulate a methodology of learning a game from rules to winning;
- learn the rules of various games from incomplete examples of legal and illegal moves, for completeness of the methodology;
- explore what strategies can be expressed using weak constraints and potential extensions to current ILASP tasks in order to cover a larger space;
- evaluate the effectiveness of the strategies learnt by ILASP against various hard-coded strategies.

1.3 CONTRIBUTIONS

There are four main contributions in this project. Firstly, we provide an exploration of ILASP's ability to learn immediate strategies, starting with choosing an appropriate representation for the game in logic (Chapter 5). Strategies such as capturing, evading capture, and controlling space are explored in Chapter 10. In order to complete the study of learning games, Chapter 9 shows ILASPs ability to learn the rules of the game after being shown some, but not all, legal and illegal moves in only a handful of board states.

We have implemented a planner that can run in two modes, *training-mode* and *tournament-mode*, which collect different types of examples to use when learning (Chapter 6). It interfaces directly with Clingo (Kaminski, 2014), an answer set solver, in an intelligent manner in order to batch evaluate many states.

We present experimentation into learning game strategies using two techniques in Chapter 10. The first is to observe the moves of the winner of a game in order to learn their strategy, the second involves having a player critique the AI whilst playing and suggesting better moves and learning solely from these counter examples.

Finally, we present an extension to the Learning from Answer Sets (LAS) framework, ILP^{deep}_{LOAS}, that is capable of learning strategies that reason into the future (Chapter 7 and Section 10.2), by not comparing examples of the same board state, but by incorporating the minimax theorem into the search of the game tree in order to find learning examples that express the strategy further down the tree. An example of a strategy that would need this technique is learning to exchange pieces on the board. Along with this extension, we present a proof of the correspondence between the set of orderings chosen by the system and the minimax theorem. The creation of the additional examples can be achieved in a *partially* automated manner given the representation of the games described in Chapter 5.

2 GAMES UNDER STUDY

2.1 ONITAMA

Onitama is a two-player, perfect information, abstract strategy game. Unlike traditional strategy games such as chess, *Onitama* has a random starting configuration. Additionally, the players' possible moves are determined by *move cards*, which vary from game to game.

Each player has two types of pawns: one *master* (♔) and four *student* (♙) pawns. The master starts on the respective players' *temple* squares.

2.1.1 RULES

OBJECTIVE

There are two winning conditions in the game, achieved either by capturing the opponent's master pawn, or by moving one's master pawn to the opponent's *temple* square.

SETUP

The game is played on a 5-by-5 grid, with each player starting with one master and four student pawns.

Five *move cards* are then drawn and each player is given two, with the last placed in the middle.

Remark. The centre card denotes who will start by means of a coloured icon on the card; in diagrams I use the side of the board to denote this. The right side of the board is blue, the left is red, i. e. the card is always on the players' right. In Figure 2.1 the blue player is starting.

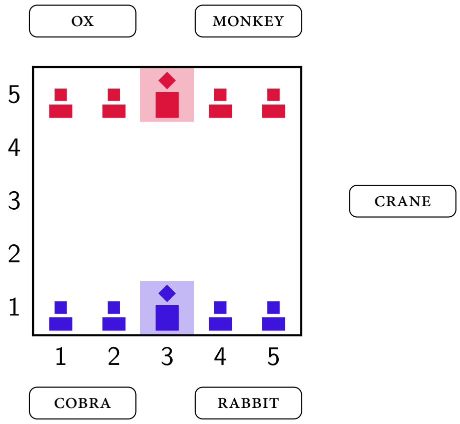


Figure 2.1. Onitama setup configuration, with the cards OX, MONKEY, CRANE, RABBIT, COBRA.

MOVING

Move cards show how a piece can move; a player may move any of their pieces according to the cards. A pawn may jump over other pawns and a player may capture an opponent's pawn only by moving on to the space that it occupies. Once a *move card* has been used it is swapped with the *move card* in the centre.

A move card shows a 5×5 grid, a central dark grey square representing the pawn's starting location, lighter grey squares representing valid moves for that pawn (these are relative to the start location). Example 2.1.1 shows all valid moves from the RABBIT card.

Remark. The location of a pawn is denoted by the pawn's symbol followed by a tuple $(Row, Column)$ where $(1,1)$ is the bottom-left, e. g. ♔ $(2,3)$ means the red master at row 2, column 3, and ♙ $(5,5)$ means a blue student at row 5, column 5 (i. e. the top-right corner).

Example 2.1.1. Figure 2.2 below shows the use of the rabbit card on the blue player's turn. It depicts several key points:

- The *move card* is being used by both ♙ and ♔ pawns
- Pawns may jump over pawns, e. g. ♔ $(3,1)$ jumping over ♙ $(3,2)$
- Pawns cannot move to a position occupied by a pawn of the same team, e. g. ♙ $(2,2)$ cannot move to $(2,4)$
- Pawns cannot move off the board, nor does it wrap, e. g. ♔ $(3,1)$ and ♙ $(2,4)$
- Pawns can capture opponent pawns, e. g. ♔ $(3,1)$ captures ♙ $(4,2)$

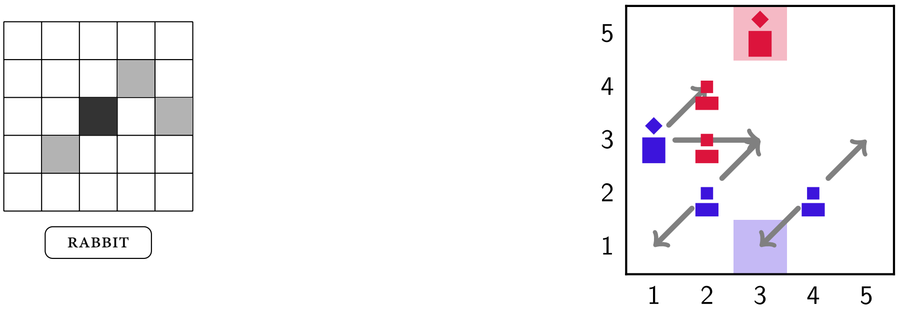


Figure 2.2. Use of the rabbit card

Example 2.1.2. Figure 2.3 shows the rabbit card being used by the red player. The only point of note here is that the card has been rotated 180° to face the red player.

2.1.2 THREE CARD VARIATION

A game of Onitama normally uses five cards, offering players a maximum of 8 moves per pawn to choose from, and allowing players to selectively withhold cards from their

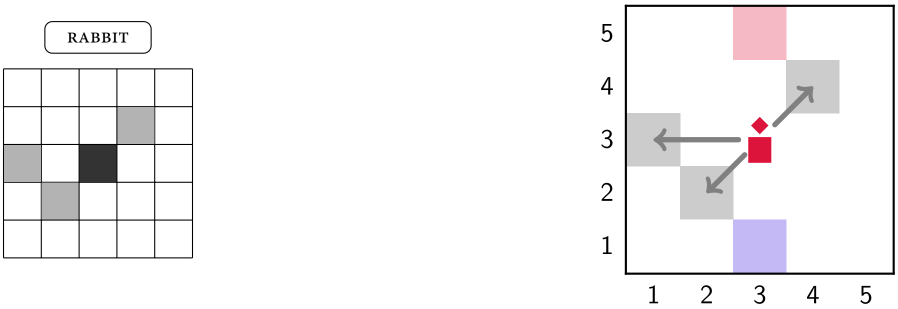


Figure 2.3. Use of the rabbit card with the opposite player

opponent. This creates a much larger state space and invites highly complex strategies. To simplify the problem whilst learning, I used just three cards; one per player and a centre card. It follows that players can now deterministically calculate the cards they will receive and therefore have certainty over which moves will be available to them in the future.

2.1.3 EXAMPLE GAME

Games of Onitama can be expressed in a modified Portable Game Notation (PGN) (Edwards, 1994). A halfmove is denoted in a similar way to a pawn's location: the pawn's symbol, starting and ending location, and the card name used. A full move is denoted by the move number, followed by the halfmove of each player. A capture is denoted using a × between the starting and ending locations.

Example 2.1.3. The following is a short example of a full game of Onitama, with the board states depicted in Figure 2.8. The cards used in this game can be found in Figure 2.4.

1. ♙ (1,2)(2,2) MOUSE ♙ (5,4)(4,4) BOAR
2. ♔ (1,3)(2,3) PANDA ♙ (5,5)(4,5) MOUSE
3. ♙ (1,4)(2,4) BOAR ♙ (4,4)(3,3) PANDA
4. ♔ (2,3) × (3,3) MOUSE ♙ (5,2)(4,2) BOAR
5. ♙ (1,5)(2,5) PANDA ♙ (4,5)(4,4) MOUSE
6. ♔ (3,3)(3,4) BOAR ♙ (4,4) × (3,4) PANDA

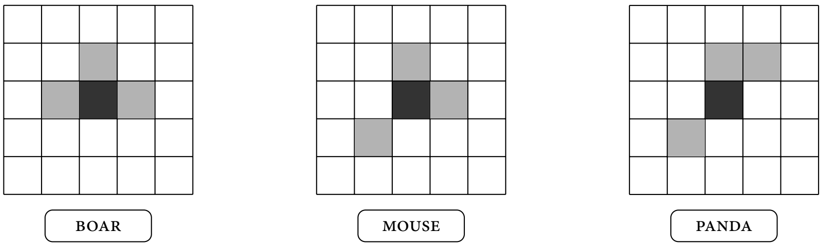
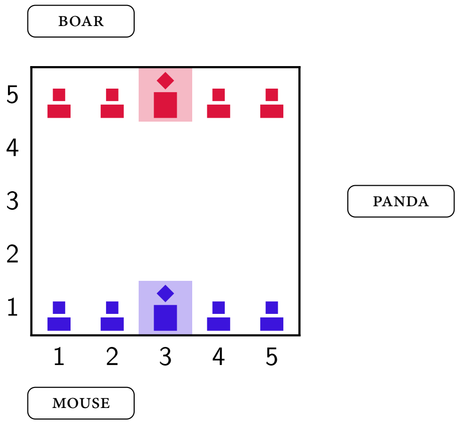


Figure 2.4. Move Cards featured in the Game



(a) Initial Board

Figure 2.5. Example game following the moves from Example 2.1.3

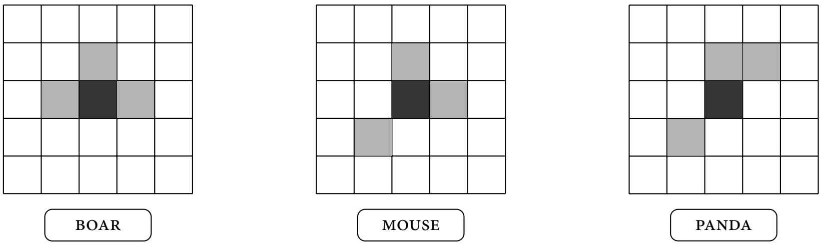
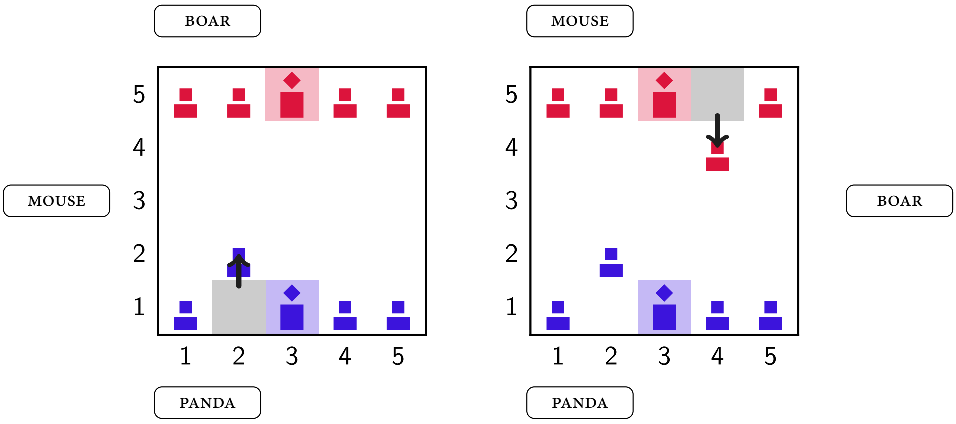
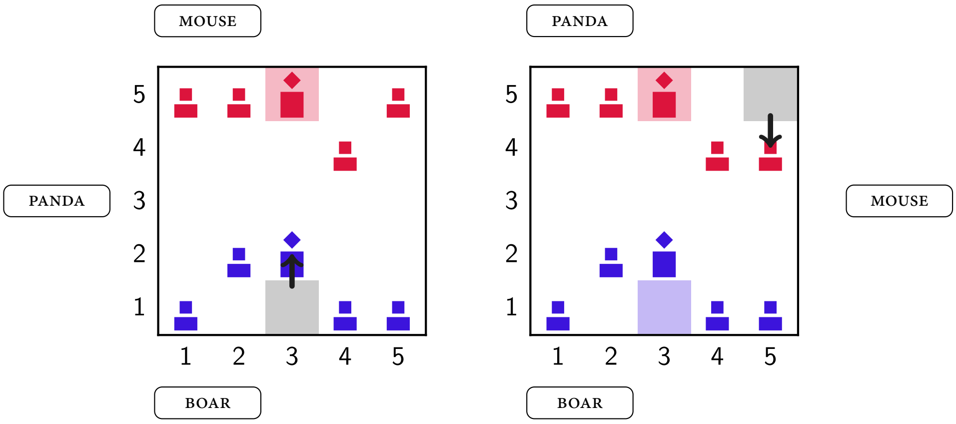


Figure 2.4. Move Cards featured in the Game (repeated from page 10)



(a) 1. ♙ $(1,2)(2,2)$ MOUSE **(b)** 1. ♙ $(5,4)(4,4)$ BOAR



(c) 2. ♔ $(1,3)(2,3)$ PANDA **(d)** 2. ♙ $(5,5)(4,5)$ MOUSE

Figure 2.6. Example game following the moves from Example 2.1.3 (cont.)

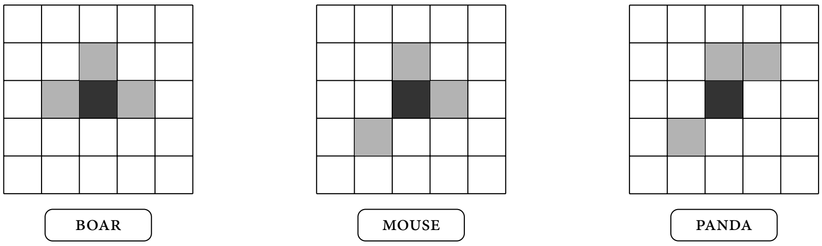
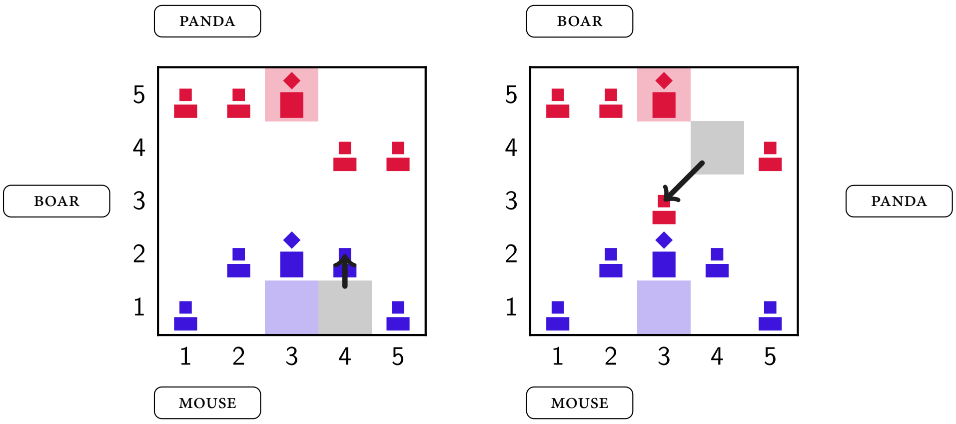
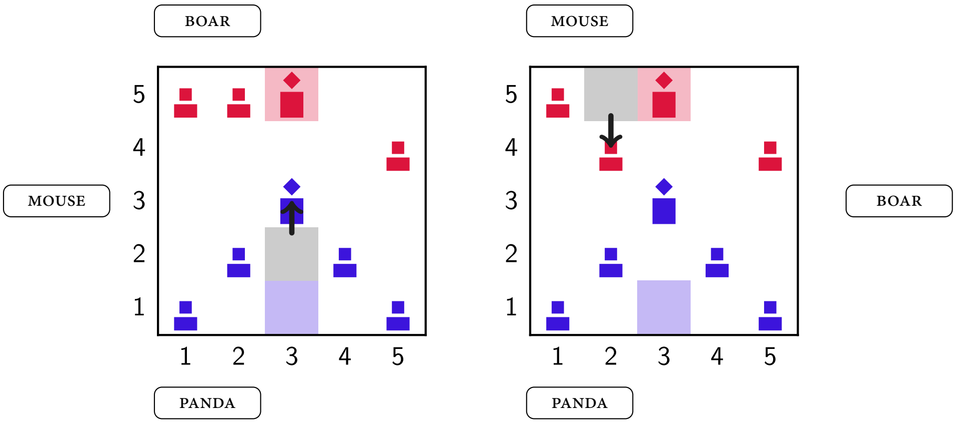


Figure 2.4. Move Cards featured in the Game (repeated from page 10)



(a) 3. ♙ $(1,4)(2,4)$ BOAR **(b)** 3. ♙ $(4,4)(3,3)$ PANDA



(c) 4. ♙ $(2,3) \times (3,3)$ MOUSE **(d)** 4. ♙ $(5,2)(4,2)$ BOAR

Figure 2.7. Example game following the moves from Example 2.1.3 (cont.)

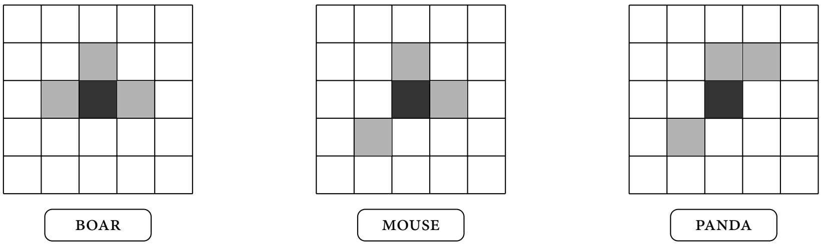
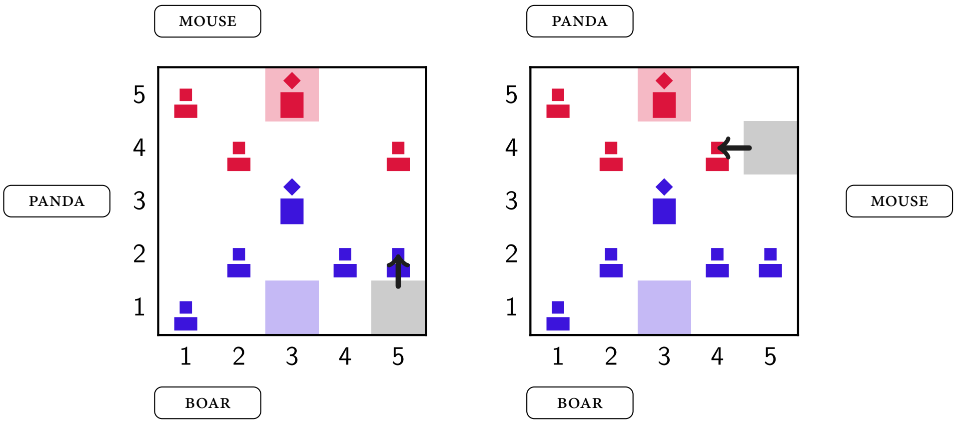
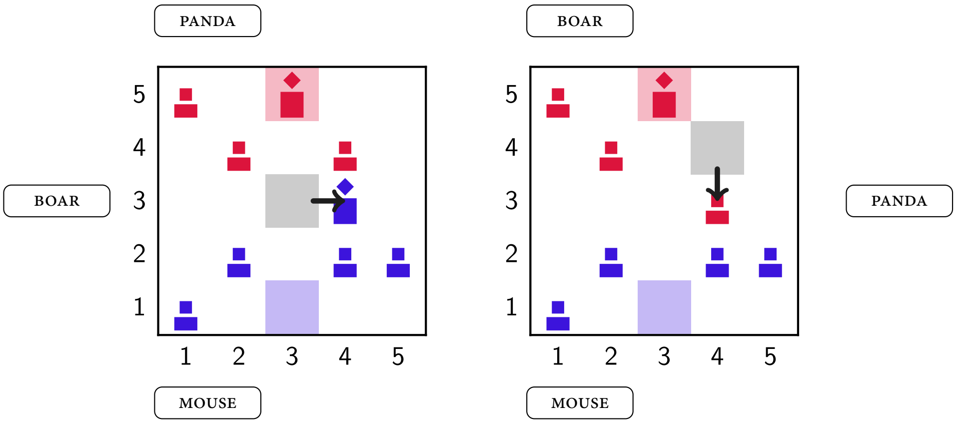


Figure 2.4. Move Cards featured in the Game (repeated from page 10)



(a) 5. ♙ (1,5)(2,5) PANDA **(b)** 5. ♙ (4,5)(4,4) MOUSE



(c) 6. ♙ (3,3)(3,4) BOAR **(d)** 6. ♙ (4,4) × (3,4) BOAR

Figure 2.8. Example game following the moves from Example 2.1.3 (cont.)

2.2 FIVE FIELD KONO

Five Field Kono is a checkers-like strategy game originating in Korea. Two players try to block and out manoeuvre each other in order to occupy the starting spaces of their opponent. All counters move in the same fashion, diagonally, and there are no captures in the game.

In this report I use Five Field Kono to demonstrate the techniques used throughout in the setting of a different game.

2.2.1 RULES

MOVES

All counters can move one space along any diagonal, but cannot jump pieces and cannot occupy the same space as another counter, as illustrated in Figure 2.9.

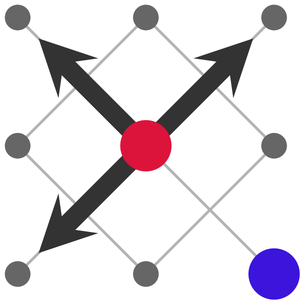
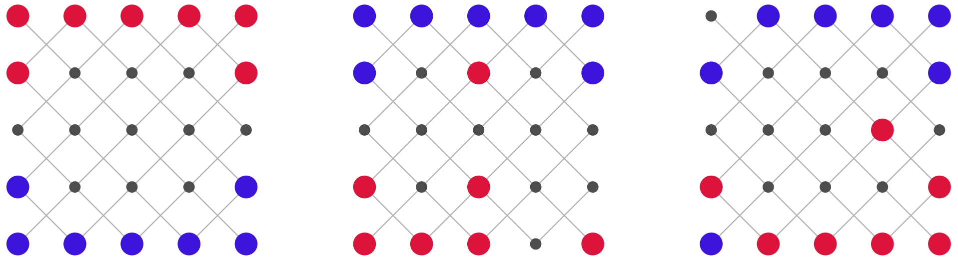


Figure 2.9. Legal moves for the red player

OBJECTIVE

A player is trying to move from their starting position to their opponent's starting position before their opponent does the same. If a player leaves counters in their starting position they count in their opponent's favour, thus their opponent only has to occupy the vacated spaces. A player must have at least one of their counters in their opponent's starting positions in order to win. Figure 2.10 shows the starting and some winning conditions of the game.



(a) Initial state of the board

(b) Winning state for blue by occupying all of red's initial spaces

(c) Winning state for red by occupying all of blue's *vacated* initial spaces

Figure 2.10. The initial state of the game of Five Field Kono alongside some winning states

2.3 CROSS-DOT GAME

The Cross-Dot game is a variation of Tic-Tac-Toe which I have taken from the literature (Zhang and Thielscher, 2015). The Cross-Dot game, another name for an m–k game, where players aim to get k boxes in a row marked with their player symbol (from a possible m boxes).

2.3.1 RULES

The rules of the game are very straight-forward, on your turn you may mark any empty box with your marker ($\times$ or $\cdot$). The board is initially completely empty, and the game ends when one of the players achieves k contiguous boxes for the win, or when all of the boxes are filled. The × player starts the game.

□ □ □ □ □ □

(a) Initial state of the board

□ ☒ ⊡ □ □ □

(b) Example of some moves

☒ ☒ ⊡ □ □ □

(c) A winning state for ×

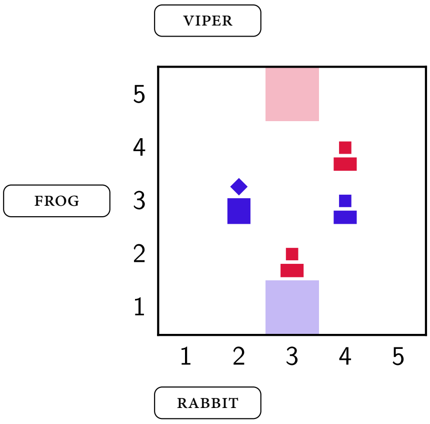
Figure 2.11. Some example states of the Cross-Dot Game ($m = 6, k = 2$)

3 BACKGROUND

3.1 NORMAL LOGIC PROGRAMS

Normal logic programs extend definite logic programs to include the notion of Negation as Failure (NAF). Negation as Failure, denoted by the operator not, differs from classical negation ($\neg$). Informally, not p means that it cannot be proven that p holds. Below is an example of NAF to illustrate its semantics.

Example 3.1.1. Below is an example game state of *Onitama* in which the blue player has just won.



$$\left\{\begin{array}{l} \mathit{master\_on\_board}(\mathit{blue}) \\ \mathit{students\_on\_board}(\mathit{blue}, 1) \\ \mathit{students\_on\_board}(\mathit{red}, 2) \end{array}\right.$$

On the right is a set of predicates that we know to be facts about the board. Below is a rule about the *master* being captured.

$$\mathit{master\_captured}(P) \leftarrow \mathit{player}(P), \text{not } \mathit{master\_on\_board}(P).$$

From this rule we can deduce that $\mathit{master\_captured}(\mathit{red})$, because it cannot be shown that $\mathit{master\_on\_board}(\mathit{red})$ is true. However, $\mathit{master\_captured}(\mathit{blue})$ cannot be deduced because, by the set of facts above, the blue master is on the board.

3.1.1 SYNTAX

Logic programs are constructed from *terms*, which consist of variables, constants, and n-ary function symbols, *atoms*, which are n-ary predicates defined over terms, and *literals* and *clauses* (defined below).

Definition 3.1.1 (literal). A *literal* is an expression of the form A, a *positive* literal, or not A, a *negative* literal, where A is an atom.

Definition 3.1.2 (rule). *Clauses*, or *rules*, are formulæ of the form

$$h \leftarrow b_1, \ldots, b_m, \text{not } c_1, \ldots, \text{not } c_n \tag{r}$$

where h, each $b_i, i \in [1, m]$ and $c_j, j \in [1, n]$ are atoms, $m \geq 0, n \geq 0$.

$\mathit{head}(r) = h$, is the head of the rule, $\mathit{body}^+(r) = \{b_1, \ldots, b_m\}$ are the positive body literals, and $\mathit{body}^-(r) = \{c_1, \ldots, c_n\}$ are the negative body literals.

A *fact* is a rule, r, with no body literals, i. e. $body^{+}(r) \cup body^{-}(r) = \emptyset$. A *constraint*, or *hard constraint*, is a rule, r, with no head, i. e. $head(r) = \bot$.

A *ground* rule is one which contains no variables, i. e. all terms that appear in the clause are constants. The set of ground rules created by substituting all variables with all combinations of atoms from the language is known as the *ground instances*. For example, the ground instances[1] of $master\_on\_board(Player) \leftarrow location(pawn(master, Player), Cell)$ are:

$$\begin{cases} master\_on\_board(red) \leftarrow player(red), location(pawn(master, red), cell(1,1)) \\ master\_on\_board(blue) \leftarrow player(blue), location(pawn(master, blue), cell(1,1)) \\ master\_on\_board(red) \leftarrow player(red), location(pawn(master, red), cell(1,2)) \\ master\_on\_board(blue) \leftarrow player(blue), location(pawn(master, blue), cell(1,2)) \\ \dots \\ master\_on\_board(red) \leftarrow player(red), location(pawn(master, red), cell(3,3)) \\ master\_on\_board(blue) \leftarrow player(blue), location(pawn(master, blue), cell(3,3)) \\ \dots \\ master\_on\_board(red) \leftarrow player(red), location(pawn(master, red), cell(5,5)) \\ master\_on\_board(blue) \leftarrow player(blue), location(pawn(master, blue), cell(5,5)) \end{cases}$$

Definition 3.1.3 (logic program). A *logic program* is a set of rules. The *grounding* of the logic program is the union of the sets of ground instances of each rule.

Example 3.1.2. Let the following *normal logic program*[2], $\Pi_{location}$, represent where pawns are on the board.

$$location(pawn(master, red), cell(2,1)). \tag{3.1}$$

$$location(pawn(master, blue), cell(3,2)). \tag{3.2}$$

$$\begin{aligned} next(&location(pawn(Rank, Player), Cell)) \\ &\leftarrow location(pawn(Rank, Player), Cell), \\ &\quad \text{not } does(\_, move(\_, Cell, \_)), \\ &\quad \text{not } does(Player, move(Cell, \_, \_)). \end{aligned} \tag{3.3}$$

$$\begin{aligned} next(&location(pawn(Rank, Player), To)) \\ &\leftarrow location(pawn(Rank, Player), From), \\ &\quad does(Player, move(From, To, \_)). \end{aligned} \tag{3.4}$$

$$\leftarrow location(P1, Cell, T), location(P2, Cell, T), P1 < P2. \tag{3.5}$$

Rules 3.1 and 3.2 are *facts*, they represent where the two master pawns are currently. Rules 3.3–3.4 are *normal rules*. They describe where a pawn is at the next time step. Rule 3.3 says a pawn will be in a cell if it is currently in that cell, nothing has been moved

---

[1] For ease of reading I have only included groundings that make semantic sense, i. e. not substituting the *Cell* variable with *red*, although they are a part of the set of ground instances

[2] _ is an anonymous variable, meaning we do not care about its value

there, and it has not been moved (i. e. a pawn is in the same location as before and it has not been captured or moved). When a pawn is moved rule 3.4 says that it will be in the new location at the next time step. There is a subtle difference between the use of the anonymous variables used in rule 3.3 and those in rule 3.4. In rule 3.3 they create a projection of the *does* predicate, and the meaning becomes "there is not a move that ends in this location". If the anonymous variables were actual variables[3] then even if there was a move that ended in *Cell*, you could satisfy the NAF literal with any other player, or card, for example. On the other hand, in rule 3.4 the anonymous variable can be substituted with a variable as the meaning simply means "for some card". Finally, rule 3.5 is a *constraint* saying that only one piece can be in a square at any given time.

Note. Rule 3.5 uses $<$ for inequality instead of $\neq$. This is because using $\neq$ means the constraint is symmetric, doubling its grounding. All terms are totally ordered and so these are equivalent.

Definition 3.1.4 (variable safety). A variable is *safe* within a rule iff it appears in a positive body literal. A rule is *safe* iff all of its variables are safe.

Example 3.1.3. The following rule are *not* safe.

$$opponent(pawn(Rank1, red), pawn(Rank2, blue)). \tag{3.6}$$

$$\begin{aligned} center\_card(Card) \leftarrow\ & \text{not } in\_hand(red, Card), \\ & \text{not } in\_hand(blue, Card). \end{aligned} \tag{3.7}$$

Rule 3.6 has variables ($Rank1, Rank2$) in the head of a rule with no body, therefore they appear in no positive body literal. In rule 3.7 the variable *Card* only appears in negative body literals.

3.1.2 HERBRAND MODELS

Definition 3.1.5 (Herbrand Base). The *Herbrand Base* of a program, Π, is the set of all ground atoms that can be constructed from predicates in Π and the ground terms that can be constructed using the terms and function symbols that occur in Π. It is denoted by $atoms(\Pi)$.

Definition 3.1.6 (Herbrand Interpretation). Let Π be a *definite* logic program (i. e. one without Negation as Failure), which is written in language $\mathcal{L}$. A *Herbrand Interpretation* is created by assigning every $\alpha \in atoms(\Pi)$ either *true* ($\top$) or *false* ($\bot$). The interpretation is usually written as the set of atoms that have been assigned to true, everything else is false.

Definition 3.1.7 (Herbrand Model). A *Herbrand Model* is a Herbrand Interpretation, I, which satisfies every rule in Π, that is to say for every rule $R \in \Pi$ if $body^+(R) \subseteq I$ and $body^-(R) \cap I = \emptyset$ then $head(R) \in I$.

Definition 3.1.8 (Least Herbrand Model). A *least Herbrand Model* is a Herbrand Model which has a $\subseteq$-minimal set of *true* ground atoms, which for definite logic programs is always unique. The least Herbrand Model of Π is denoted $\mathrm{M}(\Pi)$.

---

[3] You would also need to make the rule safe with types for the new variables

Example 3.1.4. Take the following logic program Π:

$$\begin{aligned} opponent(red, blue) && (3.8)\\ opponent(blue, red) && (3.9)\\ control(red) && (3.10)\\ next(control(Player)) \leftarrow control(Opp), opponent(Player, Opp). && (3.11) \end{aligned}$$

and the set $atoms(\Pi)$[4]:

$$\begin{cases} opponent(red, blue), opponent(blue, red), control(blue), control(red), \\ next(control(red)), next(control(blue)) \end{cases}$$

One possible Herbrand Interpretation is:

$$\begin{cases} opponent(blue, red), control(red), \\ next(control(red)), next(control(blue)) \end{cases}$$

However this is not a model as $opponent(red, blue)$ is not true and neither is the rule 3.11. Another Herbrand Interpretation is:

$$\begin{cases} opponent(red, blue), opponent(blue, red), \\ control(red), next(control(blue)) \end{cases}$$

this interpretation is a Herbrand Model. Further, it is a Least Herbrand Model.

3.1.3 STABLE MODEL SEMANTICS

The stable model semantics shown below is based on that presented by Gelfond and Lifschitz (1988).

Definition 3.1.9 (reduct)**.** Let Π be any ground normal logic program. Let $atoms(\Pi)$ be the Herbrand Base. Let $X \subseteq atoms(\Pi)$ be a set of atoms. The *reduct* Π^X is the set of clauses obtained from Π as follows:

(i) delete any clause in Π that has a formula $\text{not } A$ such that $A \in X$

(ii) delete all remaining formulæ of the form $\text{not } A$ in the bodies of the remaining clauses

or equivalently:

$$\Pi^X \triangleq \{head(r) \leftarrow body^+(r) \mid r \in \Pi, body^-(r) \cap X = \emptyset\} \tag{3.12}$$

X is a *stable model*, or *answer set*, of Π iff $X = \mathrm{M}(\Pi^X)$.

---

[4] For convenience, we only show the subset of the Herbrand Base that needs to be constructed, other atoms such as $next(control(control(blue)))$ would also be in the Herbrand Base

Example 3.1.5. Let Π be the grounding of a subset of the logic program from Example 3.1.2, with an extra fact denoting the ♙ master's move:

$$\Pi = \mathit{ground}\left(\left\{\begin{array}{l}
\mathit{location}(\mathit{pawn}(\mathit{master},\mathit{blue}),\mathit{cell}(3,2)).\\
\\
\mathit{next}(\mathit{location}(\mathit{pawn}(\mathit{Rank},\mathit{Player}),\mathit{Cell}))\\
\qquad\qquad \leftarrow \mathit{location}(\mathit{pawn}(\mathit{Rank},\mathit{Player}),\mathit{Cell}),\\
\qquad\qquad\quad \text{not } \mathit{does}(\_,\mathit{move}(\_,\mathit{Cell},\_)),\\
\qquad\qquad\quad \text{not } \mathit{does}(\mathit{Player},\mathit{move}(\mathit{Cell},\_,\_)).\\
\mathit{next}(\mathit{location}(\mathit{pawn}(\mathit{Rank},\mathit{Player}),\mathit{To}))\\
\qquad\qquad \leftarrow \mathit{location}(\mathit{pawn}(\mathit{Rank},\mathit{Player}),\mathit{From}),\\
\qquad\qquad\quad \mathit{does}(\mathit{Player},\mathit{move}(\mathit{From},\mathit{To},\_)).\\
\\
\mathit{does}(\mathit{blue},\mathit{move}(\mathit{cell}(3,2),\mathit{cell}(2,1),\mathit{monkey})).
\end{array}\right\}\right)$$

Let X be the following set of atoms:

$$\left\{\begin{array}{l}
\mathit{location}(\mathit{piece}(\mathit{master},\mathit{blue}),\mathit{cell}(3,2))\\
\mathit{next}(\mathit{location}(\mathit{piece}(\mathit{master},\mathit{blue}),\mathit{cell}(2,1)))\\
\mathit{does}(\mathit{blue},\mathit{move}(\mathit{cell}(3,2),\mathit{cell}(2,1),\mathit{monkey}))
\end{array}\right.$$

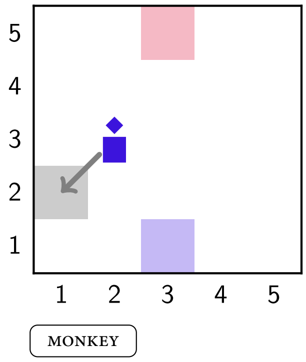


Figure 3.1. Graphical representation of X in Example 3.1.5

Constructing Π^X gives us:

$$\left\{\begin{array}{l}
\mathit{location}(\mathit{piece}(\mathit{master},\mathit{blue}),\mathit{cell}(3,2)) \\
\mathit{does}(\mathit{blue},\mathit{move}(\mathit{cell}(3,2),\mathit{cell}(2,1),\mathit{monkey})) \\
\mathit{next}(\mathit{location}(\mathit{pawn}(\mathit{master},\mathit{blue}),\mathit{cell}(3,2))) \\
\qquad\quad \leftarrow \mathit{location}(\mathit{pawn}(\mathit{master},\mathit{blue}),\mathit{cell}(3,2)), \\
\qquad\qquad \mathit{does}(\mathit{blue},\mathit{move}(\mathit{cell}(3,2),\mathit{cell}(3,2),\mathit{monkey})). \\
\mathit{next}(\mathit{location}(\mathit{pawn}(\mathit{master},\mathit{blue}),\mathit{cell}(2,1))) \\
\qquad\quad \leftarrow \mathit{location}(\mathit{pawn}(\mathit{master},\mathit{blue}),\mathit{cell}(3,2)), \\
\qquad\qquad \mathit{does}(\mathit{blue},\mathit{move}(\mathit{cell}(3,2),\mathit{cell}(2,1),\mathit{monkey})). \\
\mathit{next}(\mathit{location}(\mathit{pawn}(\mathit{master},\mathit{blue}),\mathit{cell}(3,2))) \\
\qquad\quad \leftarrow \mathit{location}(\mathit{pawn}(\mathit{master},\mathit{blue}),\mathit{cell}(2,1)), \\
\qquad\qquad \mathit{does}(\mathit{blue},\mathit{move}(\mathit{cell}(2,1),\mathit{cell}(3,2),\mathit{monkey})). \\
\mathit{next}(\mathit{location}(\mathit{pawn}(\mathit{master},\mathit{blue}),\mathit{cell}(2,1))) \\
\qquad\quad \leftarrow \mathit{location}(\mathit{pawn}(\mathit{master},\mathit{blue}),\mathit{cell}(2,1)), \\
\qquad\qquad \mathit{does}(\mathit{blue},\mathit{move}(\mathit{cell}(2,1),\mathit{cell}(2,1),\mathit{monkey})).
\end{array}\right.$$

When we compute the least Herbrand model of Π^X we get:

$$\left\{\begin{array}{l}
\mathit{location}(\mathit{pawn}(\mathit{master},\mathit{blue}),\mathit{cell}(3,2)) \\
\mathit{next}(\mathit{location}(\mathit{pawn}(\mathit{master},\mathit{blue}),\mathit{cell}(2,1))) \\
\mathit{does}(\mathit{blue},\mathit{move}(\mathit{cell}(3,2),\mathit{cell}(2,1),\mathit{monkey}))
\end{array}\right.$$

which is X, and therefore X is an *answer set*.

3.2 ANSWER SET PROGRAMMING

Answer Set Programming (ASP) is a form of declarative programming oriented towards difficult search problems (Lifschitz, 2008). It introduces new concepts such as *choice rules* and *weak constraints* (Calimeri et al., 2013), described below. Throughout this report I will be using the answer set solver Clingo.

3.2.1 CHOICE RULES

Definition 3.2.1 (choice rule). A choice rule is a *rule* with an aggregated head. It is of the form

$$l\,\{L_1,\ldots,L_h\}\,u \leftarrow B_1,\ldots,B_m \tag{3.13}$$

where $l, u \in \mathbb{N}$ are the lower and upper bound, respectively, and L_i where $i \in [1, h]$ is a literal. This aggregated head creates between l and u new rules where the head $L \in \{L_1,\ldots,L_h\}$ and the body is $B_1,\ldots,B_m$. Due to the fact there can be many ways of satisfying these conditions choices rules provide a way of generating multiple answer sets.

Example 3.2.1. This example gives the background knowledge for the moves that can be made on a turn T.

$$0\,\{does(Player, Action)\}\,1 \leftarrow legal(Player, Action), \text{not } terminal. \tag{3.14}$$

$$\begin{aligned}\leftarrow\ & does(Player, move(From1, To1, Card1)),\\ & does(Player, move(From2, To2, Card2)),\\ & neq(From1, To1, Card1) < neq(From2, To2, Card2).\end{aligned} \tag{3.15}$$

$$\leftarrow legal(\_,\_), \text{not } does(\_,\_), \text{not } terminal. \tag{3.16}$$

Rules 3.14–3.16 describe what is allowed to be a move within the game. Rule 3.14 is a *choice rule* with an lower bound of 0 and an upper bound of 1. It states that for each grounding of a valid move we can either choose to make a move or not. Rules 3.15 and 3.16 together enforce that there is at least one unique move at any given time, so long as there is a valid move that can be made. The case when there is no valid move is dealt with separately.

Remark. If Rule 3.14 was a normal rule the program would be unsatisfiable because it would force all valid moves to be chosen moves, violating the uniqueness constraint.

3.2.2 WEAK CONSTRAINTS

Weak constraints were originally introduced in Disjunctive Datalog (Buccafurri, Leone and Rullo, 1997) in order to specify integrity constraints that should be satisfied *if possible*. They were used to express optimisation ideas such as "ensure there are as few timetable clashes as possible". In this report we are trying to optimise a player's performance in a game. ,Thus, we can harness weak constraints to represent tactics in a game that build upon each other in order to describe a strategy.

Definition 3.2.2 (weak constraint). *Weak constraints* create an ordering over $\mathcal{AS}(\Pi)$, specifying which answer sets are "better" than others. Unlike hard constraints, weak constraints do not affect what is or is not in an answer set of a program Π (Law, Russo and Broda, 2015a).

A weak constraint is of the form

$$\leftarrow\!\!\!\sim b_1, \ldots, b_m, \text{not } c_1, \ldots, \text{not } c_n.[w@l, t_1, \ldots, t_o] \tag{3.17}$$

where $b_1, \ldots, b_m, c_1, \ldots, c_n$ are atoms from $\mathcal{L}_H$, w and l are terms specifying the *weight* and *level*, and $t_1, \ldots, t_o$ are terms. All variables in weak constraint 3.17 must be safe, i. e. all variables that appear in $t_1, \ldots, t_o$ must appear in the positive body literals.

When a weak constraint's body is satisfied a tuple $(w, l, t_1, \ldots, t_o)$ is generated, where t_i are the ground terms in the weak constraint. For any ASP program Π and answer set $A \in \mathcal{AS}(\Pi)$ there is a set $weak(\Pi, A)$ of tuples $(w, l, t_1, \ldots, t_o)$ corresponding to weak constraints whose body is satisfied.

Example 3.2.2. The following weak constraints are one possible way of expressing that a player wishes to win first and foremost, and capture pieces whilst maintaining their own if possible. The `goal(·,·)` predicate[5] is which denotes that a player has received some reward for meeting some game condition (e. g. winning).

$$\leftsquigarrow \text{not } goal(red, 100).[5@2] \tag{3.18}$$

$$\leftsquigarrow location(pawn(student, blue), Cell)[1@1, Cell] \tag{3.19}$$

$$\leftsquigarrow location(pawn(student, red), Cell)[-1@1, Cell] \tag{3.20}$$

Weak Constraint 3.18 says that if we cannot show that *red* has won then we incur a penalty of 5. The priority level 2 means that this constraint will be minimised first. Weak constraint 3.19 says that if we find a blue student (♙) on the board then we incur a penalty of 1. The variable *Cell* means we incur this penalty for each unique grounding of these variables, i. e. each blue student on the board. Weak constraint 3.20 says we gain a reward for each red student (♙) on the board. Both weak constraint 3.19 and 3.20 are at priority level 1 and so the penalties are added (e. g. if there were $2\times$♙ and $1\times$♙ on the board, and it is not a winning state for red, the score would be 5 at level 2, and 1 at level 1).

The semantics for weak constraints presented below are based on those in Calimeri et al. (2013) and Law, Russo and Broda (2015a). For any level $l \in \mathbb{N}$, $A \in \mathcal{AS}(\Pi)$ let

$$P_l^A = \sum_{w \in W_A^l} w$$

denote the penalty of the answer set $A \in \mathcal{AS}(\Pi)$, at level l where $W_l^A = \{w \mid (w, l, t_1, \ldots, t_o) \in weak(\Pi, A)\}$. A_1 dominates A_2 ($A_1 \succ_\Pi A_2$) iff $\exists l$ such that $P_l^{A_1} < P_l^{A_2}$ and $\forall l' > l \Rightarrow P_{l'}^{A_1} = P_{l'}^{A_2}$

Example 3.2.3. Given some program Π containing weak constraints 3.18–3.20 such that the *only* answer sets are $\mathcal{AS}(\Pi) = \{A_1, A_2\}$, let

$$\begin{aligned}
weak(\Pi, A_1) &= \{(5,2), (1,1,cell(3,2)), (1,1,cell(2,4)), (-1,1,cell(2,3))\}\\
weak(\Pi, A_2) &= \{(5,2), (1,1,cell(5,4)), (-1,1,cell(3,4))\}\\
P_{A_1}^2 &= 5\\
P_{A_1}^1 &= 1+1-1 = 1\\
P_{A_2}^2 &= 5\\
P_{A_2}^1 &= 1-1 = 0
\end{aligned}$$

therefore $A_2 \succ_\Pi A_1$ as the penalties at level 2 are equal and $P_{A_2}^1 < P_{A_1}^1$. Additionally as no answer set dominates A_2 it is *optimal*.

Remark. Clingo uses the notation :- in place of $\leftarrow$, and :~ in place of $\leftsquigarrow$.

3.3 INDUCTIVE LOGIC PROGRAMMING

The field of Inductive Logic Programming (ILP) essentially combines Machine Learning with logical knowledge representation (Muggleton et al., 2011). The definition of a inductive learning problem is given in Muggleton (1991) and Muggleton and Raedt (1994). The general setting for a inductive logic program has three languages:

---

[5] We will later see in Chapter 5 that this predicate is defined by a game description language

$\mathcal{L}_E$: The language of the examples

$\mathcal{L}_B$: The language of the background knowledge

$\mathcal{L}_H$: The language of the hypotheses

The general induction learning problem is then defined as follows: given a set of *ground, atomic* examples or observations $E^+ \subseteq \mathcal{L}_E$, a background knowledge $B \subseteq \mathcal{L}_B$, both consistent, find a hypothesis $H \subseteq \mathcal{L}_H$ such that $B \cup H \models E^+$, that is, the background knowledge together with the learnt hypothesis entail, or *cover*, the observations.

A set of negative examples $E^- \subseteq \mathcal{L}_E$ can also be given (Muggleton and Raedt, 1994), it is then required that the background knowledge together with the hypothesis is *consistent* with respect to the negative examples, $\forall e^- \in E^- \,.\, B \cup H \cup e^- \models \square$.

3.4 INDUCTIVE LEARNING OF ANSWER SET PROGRAMS

In Law, Russo and Broda (2014) the concept of ILASP was introduced. This expanded on previous work in the ILP field such as *Progol*, *Metagol* and HAIL (Muggleton, 1995; Muggleton et al., 2014, and; Ray, Broda and Russo, 2003). The previous work mainly focussed on definite and normal logic programs, whereas ILASP looks at a different class of programs; Answer Set Programs (Law, Russo and Broda, 2014). In this section I will look into the first version of ILASP. I also explore further advances, for example, learning from weak constraints (Law, Russo and Broda, 2015a) and learning from context dependent examples (Law, Russo and Broda, 2016).

The problem is structured in a similar manner to the general inductive problem. However the sets of examples, E^+, E^-, are now no longer individual atoms, but instead *partial interpretations*, meaning a problem from *Progol*, for example, could be constructed with a single example.

In this section, it is assumed that background knowledge and hypotheses are ASP programs, as defined in Section 3.2.

3.4.1 LEARNING FROM ANSWER SETS

In this section, I define the ILASP paradigm based on Law, Russo and Broda (2014).

In an ILP task the hypothesis space is defined by a language bias, which specifies how $\mathcal{L}_H$ is built.

Definition 3.4.1 (language bias). A Learning from Answer Sets (LAS) *language bias* is defined as a pair of mode declarations $M^{\text{LAS}} = \langle M_h, M_b \rangle$, where M_h and M_b are the *head* and *body mode declarations*, respectively. Both M_h and M_b are sets of literals with their arguments replaced with v_{type} and c_{type} denoting ‘variable’ and ‘constant’, respectively, where *type* is the type of the variable or constant.

Given a language bias M^{LAS}, a rule, r, of the form $L_h \leftarrow L_1, \ldots, L_m,$
$\text{not } L_{m+1}, \ldots, \text{not } L_n$ such that $n \geq m \geq 0$ is in the search space S_M^{LAS} iff

(i) either

  a) L_h is empty *or*

  b) L_h is compatible with M_h *or*

c) L_h is an aggregate $l\ \{h_1, \ldots, h_k\}\ u$ such that $0 \leq l \leq u \leq k$ and $\forall i \in [0, k]$ h_i is compatible with M_h

(ii) $\forall i \in [1, n]$ L_i is compatible with M_b

(iii) All variables are safe

Finally, each rule $r \in S_M^{\text{LAS}}$ is given a unique identifier r_{id}.

Remark. Informally, a literal is *compatible* with a mode declaration m if there exists an instance of v (resp. c) that can be replaced by a variable (resp. constant).

Example 3.4.1. Let $M^{\text{LAS}} = \langle \{goal(\mathrm{v}_{role}, \mathrm{c}_{reward})\}, \{opponent(\mathrm{v}_{role}, \mathrm{v}_{role}), temple(\mathrm{v}_{role}, \mathrm{v}_{cell}), location(pawn(\mathrm{c}_{rank}, \mathrm{v}_{role}), \mathrm{v}_{cell})\}\rangle$. Then the following rules are in S_M^{LAS}:

$$\begin{aligned} goal(Player, 100) \leftarrow\ & opponent(Player, Enemy), \\ & temple(Enemy, Temple), \\ & location(pawn(master, Player), Temple). \end{aligned}$$

$$\begin{aligned} \leftarrow\ & \text{not } opponent(Player, Enemy), \\ & temple(Enemy, Cell), \\ & location(pawn(master, Player), Cell). \end{aligned}$$

However the following is not:

$$\begin{aligned} goal(red, Reward) \leftarrow\ & opponent(red, Enemy), \\ & temple(Enemy, Cell), \\ & location(pawn(master, red), Cell). \end{aligned}$$

because there is no head mode declaration that allows the *goal* predicate with this combination of variables and constants, similarly for the body literal *opponent*, and because the second argument in *pawn* should be a variable according to the language bias.

Definition 3.4.2 (partial interpretation). A *partial interpretation* E is a pair $\langle E^{inc}, E^{exc}\rangle$ of sets of ground atoms from $\mathcal{L}_E$, called the *inclusions* and *exclusions*, respectively. An Answer Set A *extends* $\langle E^{inc}, E^{exc}\rangle$ iff $E^{inc} \subseteq A$ and $E^{exc} \cap A = \emptyset$.

A partial interpretation E is *bravely* entailed by a program Π iff there exists $A \in \mathcal{AS}(\Pi)$ such that A extends E. E is *cautiously* entailed by Π iff for every $A \in \mathcal{AS}(\Pi)$ A extends E.

Definition 3.4.3 (Learning from Answer Sets task). A *Learning from Answer Sets task* is a tuple $T = \langle B, S_M^{\text{LAS}}, E^+, E^- \rangle$ where B is the background knowledge, S_M^{LAS} is the search space defined by the language bias M^{LAS}, and E^+ and E^- are partial interpretations called, respectively, the positive and negative examples.

A hypothesis H is an inductive solution of T ($H \in ILP_{\text{LAS}}(T)$) iff:

(i) $H \subseteq S_M^{\text{LAS}}$

(ii) $\forall e^+ \in E^+\ \exists A \in \mathcal{AS}(B \cup H)$ such that A extends e^+

(iii) $\forall e^- \in E^-\ \neg\exists A \in \mathcal{AS}(B \cup H)$ such that A extends e^-

Example 3.4.2. In this example we try to learn who the start player of a game of onitama is. Recall from Section 2.1 that all the *move cards* have a icon depicting either the red or the blue player, here this is represented by `icon(·,·)`. `does(·,·)` denotes an action taken by a player, `control(·)` denotes whose turn it is, other predicates have their intuitive meaning.

Let $T = \langle B, S^{\text{LAS}}_{M}, E^{+}, E^{-}\rangle$ be a learning task where B is the following logic program:[6] [7]

$$in\_play(panda; mouse; fox). \tag{3.21}$$
$$in\_hand(red, panda). \tag{3.22}$$
$$in\_hand(blue, mouse). \tag{3.23}$$
$$does(red, move(cell(4,2), cell(3,2), panda)). \tag{3.24}$$
$$icon(panda, red). \tag{3.25}$$
$$icon(mouse, blue). \tag{3.26}$$
$$icon(fox, red). \tag{3.27}$$
$$icon(viper, red). \tag{3.28}$$

$$control(Player) \leftarrow init(control(Player)), \text{not } control(\_). \tag{3.29}$$
$$next(control(red)) \leftarrow control(blue). \tag{3.30}$$
$$next(control(blue)) \leftarrow control(red). \tag{3.31}$$

$$next(in\_hand(Player, Card)) \leftarrow in\_hand(Player, Card), \text{not } control(Player). \tag{3.32}$$
$$next(in\_hand(Player, Card)) \leftarrow center\_card(Card), control(Player). \tag{3.33}$$

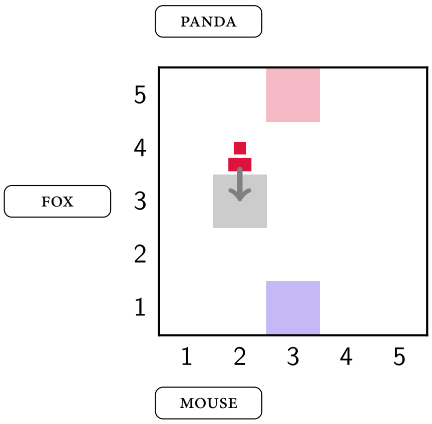


Figure 3.2. Simplified graphical representation of the background facts of B in Example 3.4.2

$$M^{\text{LAS}} = \left\langle \begin{array}{l} \{init(control(\mathrm{v}_{role})), center\_card(\mathrm{v}_{card})\}, \\ \{in\_play(\mathrm{v}_{card}), in\_hand(\mathrm{c}_{role}, \mathrm{v}_{card}), icon(\mathrm{v}_{card}, \mathrm{v}_{role}), center\_card(\mathrm{v}_{card})\} \end{array} \right\rangle$$

---

[6] Some types have been omitted for brevity

[7] ; is used to abbreviate atoms e. g. $a(b;c). \equiv a(b).a(c).$

Then the following set of positive examples say that on the second turn the red player must be in possession of the FOX card, and that neither MOUSE nor PANDA are the initial centre card.

$$E^+ = \{\langle\{next(in\_hand(red, fox))\}, \{center\_card(mouse), center\_card(panda)\}\rangle\}$$

Additionally, the negative examples below say that in no answer set can both players start, and that the VIPER is never the centre card.

$$E^- = \{\langle\{init(control(red)), init(control(blue))\}, \emptyset\rangle, \langle\{center\_card(viper)\}, \emptyset\rangle\}$$

The learnt hypothesis H is:

$$init(control(Player)) \leftarrow center\_card(Card), icon(Card, Player). \tag{3.34}$$

$$\begin{aligned} center\_card(Card) \leftarrow\ & in\_play(Card), \\ & \text{not } in\_hand(blue, Card), \\ & \text{not } in\_hand(red, Card). \end{aligned} \tag{3.35}$$

Rule 3.34 says that the first player is given by the player icon on the centre card. Rule 3.35 says that the centre card is the card in play that is in neither player's hand.

It is possible that several hypotheses will cover the examples, ILASP will choose one of the shortest length, defined below.

Definition 3.4.4 (hypothesis length). Given a hypothesis H, the length of the hypothesis $|H|$ is the number of literals in H^D where H^D is obtained from H by replacing all the aggregates by their disjunctive normal form. For example, hypothesis H above (rules 3.34 and 3.35) is length 7.

3.4.2 LEARNING WEAK CONSTRAINTS

So far we have seen that answer sets can represent states of the board (i. e. Figure 3.2) and that multiple moves can generate several answer sets using choice rules (Definition 3.2.1), which can be used to represent small expanded sections of a game tree. When learning strategies we need some notion of preference over these answer sets, and by extension the moves. In commonly used machine learning methods for playing games, such as reinforcement learning, a *state-action value function* can be used to estimate the reward of taking an action in a certain state. Answer Set Programming provides us with weak constraints (Definition 3.2.2) which can assign penalties[8] to answer sets, acting as a form of state-action value function. Law, Russo and Broda (2015a) introduces LOAS, a system that extends LAS to learn these weak constraints.

In previous ILP systems such as TILDE (Blockeel and De Raedt, 1998), preferential learning was approached as a classification problem (Dastani et al., 2005), with each labelled as 'good' or 'bad'. The drawback is that it offers no relative preference between two examples classified as 'good' (or 'bad').

In order to learn weak constraints we need to extend the notion of mode declaration to be able to generate a language bias that includes weak constraints. Therefore two new mode declarations are added: M_o which specifies what can appear in the body of a weak

---

[8] or negative penalties, i. e. rewards

constraint, and M_w which specifies what can appear as a weight. We also give $l_{max} \in \mathbb{N}$ as the maximum level that can appear in H.

Weak constraints are the crux of learning the strategies of a player. They encode why a player perceives one move to be better than the other possible moves. We use the weight of the weak constraint as a penalty and the best move is the answer set that receives the lowest penalty.

Definitions 3.4.5, 3.4.6 and 3.4.7 are from Law, Russo and Broda (2015a), and extend definitions seen in previous sections.

Definition 3.4.5 (language bias). This definition extends Definition 3.4.1. In a LOAS task the language bias is defined by the mode declaration $M = \langle M_h, M_b, M_o, M_w, l_{max} \rangle$. The search space S_M is the set of rules such that $r \in S_M$ satisfies one of the following conditions:

(i) $r \in S_M^{\text{LAS}}$, where $M^{\text{LAS}} = \langle M_h, M_b \rangle$

(ii) r is a weak constraint $:\sim b_1, \ldots, b_m, \text{not } c_1, \ldots, \text{not } c_n.[w@l, t_1, \ldots, t_o]$ such that $w \in M_w$, $l \in [0, l_{max}]$, $t_1, \ldots, t_o$ is the set of terms in $body^+(r) \cup body^-(r)$ and each b_i, c_j is compatible with M_o where $i \in [0, m], j \in [0, n]$.

Remark. Because weak constraints do not alter what is contained in an answer set one could always remove the weak constraints from a hypothesis and it would be more optimal and still cover the examples. To solve this we need to introduce the notion of preferred examples to the learning task, the weak constraints are then used to cover this type of example.

Definition 3.4.6 (ordering example). An *ordering example* is a pair $o = \langle e_1, e_2 \rangle$ where e_1, e_2 are partial interpretations. An ASP program Π *bravely* respects o iff $\exists A_1, A_2 \in \mathcal{AS}(\Pi)$ such that A_1 extends e_1, A_2 extends e_2 and $A_1 \succ_\Pi A_2$. An ASP program *cautiously* respects o iff $\forall A_1, A_2 \in \mathcal{AS}(\Pi)$ such that A_1 extends e_1, A_2 extends e_2, it is the case that $A_1 \succ_\Pi A_2$.

Definition 3.4.7 (Learning from Ordered Answer Sets task). A Learning from Ordered Answer Sets task is a tuple $T = \langle B, S_M, E^+, E^-, O^b, O^c \rangle$, where

(i) B is the background knowledge,

(ii) S_M is the search space defined by the mode declaration,

(iii) $M = \langle M_h, M_b, M_o, M_w, l_{max} \rangle$,

(iv) E^+, E^- are the positive and negative examples, respectively,

(v) O^b, O^c are sets of ordering examples over E^+ called the brave and cautious orderings, respectively.

A hypothesis $H \subseteq S_M$ is in $ILP_{\text{LOAS}}(T)$, the inductive solutions of T, iff:

(i) $H' \in ILP_{\text{LAS}}(\langle B, S_M^{\text{LAS}}, E^+, E^- \rangle)$, where H' is the subset of H with no weak constraints, $M^{\text{LAS}} = \langle M_h, M_b \rangle$.

(ii) $\forall o \in O^b$, such that $B \cup H$ bravely respects o

(iii) $\forall o \in O^c$, such that $B \cup H$ cautiously respects o

3.4.3 CONTEXT DEPENDENT EXAMPLES

When learning strategies and rules of a game it is very likely that one board state will not encapsulate the full rules or nuances of the strategy. For example, preferring to win than capture a student pawn in *Onitama* cannot be inferred from one state in which it is possible to win but not capture, or for that matter one in which it is not possible to win! Therefore, when providing an example we wish to couple it to the state of the board when the move was being made (the context of the move).

To take this into account I present extensions of Definitions 3.4.2, 3.4.6, and 3.4.7, in accordance with Law, Russo and Broda (2016), where the concept of coupling each example with its context was introduced.

Definition 3.4.8 (Context Dependent Partial Interpretation)**.** This definition extends Definition 3.4.2. A Context Dependent Partial Interpretation (CDPI) is a tuple $\langle e, C\rangle$ where e is a partial interpretation and C is an ASP program without weak constraints, called the *context*.

Definition 3.4.9 (Context Dependent Ordering Example)**.** This definition extends Definition 3.4.6. The only difference is that the answer sets include the contexts of their respective CDPI. A Context Dependent Ordering Example (CDOE) o is a tuple of CDPIs $\langle\langle e_1, C_1\rangle, \langle e_2, C_2\rangle\rangle$. An ASP program Π *bravely* respects o iff $\exists A_1 \in \mathcal{AS}(\Pi \cup C_1), A_2 \in \mathcal{AS}(\Pi \cup C_2)$ such that A_1 extends e_1, A_2 extends e_2 and $A_1 \succ_\Pi A_2$. An ASP program *cautiously* respects o iff $\forall A_1 \in \mathcal{AS}(\Pi \cup C_1), A_2 \in \mathcal{AS}(\Pi \cup C_2)$ such that A_1 extends e_1, A_2 extends e_2, it is the case that $A_1 \succ_\Pi A_2$.

Definition 3.4.10 (Context Dependent LOAS task)**.** This definition extends Definition 3.4.7. A Context Dependent Learning from Ordered Answer Sets task is a tuple $T = \langle B, S_M, E^+, E^-, O^b, O^c\rangle$ where the examples are now CDPIs and the orderings are CDOEs. A hypothesis H is an inductive solution of T, $H \in ILP^{context}_{\text{LOAS}}(T)$ iff:

(i) $H \subseteq S_M$,

(ii) $\forall\langle e, C\rangle \in E^+, \exists A \in \mathcal{AS}(B \cup C \cup H)$ such that A extends e,

(iii) $\forall\langle e, C\rangle \in E^-, \neg\exists A \in \mathcal{AS}(B \cup C \cup H)$ such that A extends e,

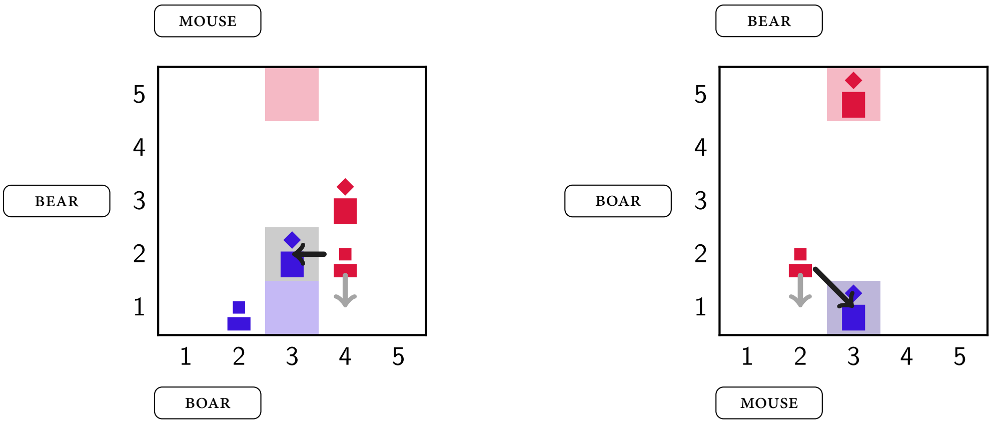
(iv) $\forall o \in O^b$, such that $B \cup H$ bravely respects o,

(v) $\forall o \in O^c$, such that $B \cup H$ cautiously respects o.

Example 3.4.3. Let $T = \langle B, S_M, E^+, E^-, O^b, O^c \rangle$ be a $ILP^{context}_{LOAS}$ task where B is the background knowledge from Logic Program A.1, together with the facts $\{in\_play(mouse; boar; bear); control(red)\}$,

$$
\begin{aligned}
M &= \langle \emptyset, \emptyset, \{goal(\mathrm{c}_{role}, 100)\}, \{-1, 1\}, 2\rangle, \\
C_1 &= \begin{cases} location(pawn(student, blue), cell(1,2)). \\ location(pawn(master, blue), cell(2,3)). \\ location(pawn(student, red), cell(2,4)). \\ location(pawn(master, red), cell(3,4)). \\ in\_hand(red, mouse). \\ in\_hand(blue, boar). \end{cases} \\
C_2 &= \begin{cases} location(pawn(master, blue), cell(1,3)). \\ location(pawn(student, red), cell(2,2)). \\ location(pawn(master, red), cell(5,3)). \\ in\_hand(red, bear). \\ in\_hand(blue, mouse). \end{cases} \\
e_1 &= \langle \{does(red, move(cell(2,4), cell(2,3), mouse))\}, \emptyset, C_1 \rangle \\
e_2 &= \langle \{does(red, move(cell(2,4), cell(1,4), mouse))\}, \emptyset, C_1 \rangle \\
e_3 &= \langle \{does(red, move(cell(2,2), cell(1,3), bear))\}, \emptyset, C_2 \rangle \\
e_4 &= \langle \{does(red, move(cell(2,2), cell(1,2), bear))\}, \emptyset, C_2 \rangle \\
E^+ &= \{e_1, e_2, e_3, e_4\}, \qquad\qquad E^- = \emptyset, \\
O^b &= \{\langle e_1, e_2\rangle, \langle e_3, e_4\rangle\}, \qquad\qquad O^c = \emptyset.
\end{aligned}
$$

The examples are shown in Figure 3.3.



(a) State of the board C_1 showing examples e_1 and e_2

(b) State of the board C_2 showing examples e_3 and e_4

Figure 3.3. Examples e_1, e_2, e_3, e_4 in their respective contexts with the preferred action shown by the highlighted move.

The hypothesis $H = \{\leftsquigarrow \text{not } goal(red, 100).[1@1]\}$ is the inductive hypothesis for T, alternatively you could have $H' = \{\leftsquigarrow goal(red, 100).[-1@1]\}$.

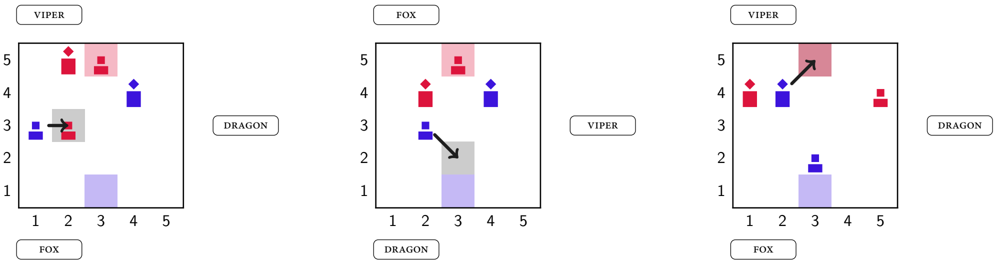
3.4.4 LEARNING FROM NOISY EXAMPLES

In complex games a player's strategy will not always be strictly enforced; the player might deviate or make a mistake. When this happens the moves in the game will not all conform to a consistent strategy that can be learnt. To account for this we use a feature of ILASP called *noise*. Examples or ordering examples can be assigned a *noise weight*, $w \in \mathbb{N}$, which will be accounted for when learning a hypothesis.

Normally, the length of a hypothesis (Definition 3.4.4) is used to choose the optimal hypothesis for a task. When noise is involved the heuristic is $|H| + W$, where W is the sum of all noise weights of the *uncovered* examples.

Example 3.4.4. Let us assume Alan is using the strategy "win if possible, otherwise capture a piece, otherwise move randomly". Some example moves may look like those in Figure 3.4a–3.4c. However in Figure 3.4d we can see a noisy example from earlier in the game.

Recall $\Pi_{location}$ from Example 3.1.2: B is created by removing the facts from $\Pi_{location}$; $M = \langle \emptyset, \emptyset, \{next(location(pawn(\mathrm{v}_{rank}, \mathrm{c}_{role}), \mathrm{v}_{cell})), goal(\mathrm{c}_{role}, 100)\}, \{-1, 1\}, 2\rangle$; E^+ is the set of examples created with empty inclusions and exclusions, using the four states in Figure 3.4a–3.4d and all possible moves from each board position as the context; and O^b is the set of orderings created by comparing the moves shown in Figure 3.4a–3.4d to all other moves from that state.

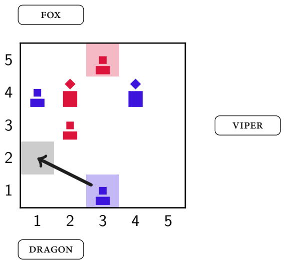


(a) Alan captures one of Betty's student pawns

(b) Alan moves randomly as is unable to win or capture

(c) Alan wins the game (some turns later)

Figure 3.4. Some of Alan's moves from a particular game against Betty



(d) Alan ignores the student pawn on $(3, 2)$ in favour of moving a piece up the board.

Figure 3.4. Some of Alan's moves from a particular game against Betty (cont.)

ILASP is able to learn the following strategy, treating Figure 3.4d as noise:

$$\leftsquigarrow goal(blue, 100).[-1@1] \tag{3.36}$$

$$\leftsquigarrow next(location(pawn(Rank, red), Cell).[1@1, Rank, Cell] \tag{3.37}$$

Weak constraint 3.36 provides a reward for winning, and weak constraint 3.37 says minimise the number of red pawns on the board. This is thus akin to Alan's strategy.

Remark. As there is no example in which there is a choice between winning and capturing they have been learnt at the same level.

3.4.5 CONSTRAINING THE HYPOTHESIS SPACE WITH BIAS CONSTRAINTS

When the mode bias grows with numerous heads and bodies the number of potential rules in the hypothesis space becomes vast. This effect is magnified when there are a lot of common variables and the maximum number of variables is greater than 7-8.[9] A large proportion of these rules can be considered nonsense semantically, or represent something you know is uninteresting (e. g. you may wish for a series of variables of the same type to be distinct). In order to filter these rules from the hypothesis space *bias constraints* are used. They are a set of meta rules that are applied to the hypothesis space, clause by clause (i. e. not globally), a series of predicates have been defined to aid in filtering. They can be found in Table 3.1.

LOGIC	DESCRIPTION
`head(·)`	The predicate is in the head of the rule
`body(·)`	The predicate is in the body of the rule
`constraint`	The rule is a (hard) constraint
`weak_constraint`	The rule is a weak constraint
`weight(·)`	The weight of the weak constraint, variables are made lower case
`term(·,·)`	The index and lower case version of a term in the weak constraint
`naf(·)`	The literal is a NAF literal

Table 3.1. Wrapper predicates to indicate structural elements of rules in the hypothesis space

Example 3.4.5. Take the following mode declarations:

$$M = \left\langle \begin{array}{l} \{does(\mathrm{v}_{role}, move((\mathrm{v}_{index}, \mathrm{v}_{index}), (\mathrm{v}_{index}, \mathrm{v}_{index})))\}, \\ \{cell(\mathrm{v}_{index}, \mathrm{v}_{index}), role(\mathrm{v}_{role})\} \end{array} \right\rangle$$

---

[9] Based on empirical evidence of running various ILASP tasks with `-s`

ILASP produces 1062 rules from this mode bias. This can be made much smaller using the following bias constraints.

$$\leftarrow \mathit{constraint}. \tag{3.38}$$

$$\leftarrow \mathit{body}(\mathit{naf}(\mathit{cell}(\_,\_))). \tag{3.39}$$

$$\leftarrow \mathit{body}(\mathit{naf}(\mathit{role}(\_))). \tag{3.40}$$

$$\leftarrow \mathit{head}(\mathit{does}(\_,\mathit{move}((V,V),(\_,\_)))). \tag{3.41}$$

$$\leftarrow \mathit{head}(\mathit{does}(\_,\mathit{move}((\_,\_),(V,V)))). \tag{3.42}$$

$$\leftarrow \mathit{head}(\mathit{does}(\_,\mathit{move}((V,\_),(V,\_)))). \tag{3.43}$$

$$\leftarrow \mathit{head}(\mathit{does}(\_,\mathit{move}((\_,V),(\_,V)))). \tag{3.44}$$

$$\leftarrow \mathit{head}(\mathit{does}(\_,\mathit{move}((V,V),(\_,\_)))). \tag{3.45}$$

$$\leftarrow \mathit{head}(\mathit{does}(\_,\mathit{move}((V,V),(\_,\_)))). \tag{3.46}$$

$$\leftarrow \mathit{head}(\mathit{does}(\_,\mathit{move}((V,\_),(\_,V)))). \tag{3.47}$$

$$\leftarrow \mathit{head}(\mathit{does}(\_,\mathit{move}((\_,V),(V,\_)))). \tag{3.48}$$

Equation 3.38 means that there can be no constraints.[10] Equation 3.39–3.40 express that we do not want the types to be NAF literals.[11] Equation 3.41–3.48 express that no two indices in the head should be forced to be identical.

With the addition of these bias constraints into the program the search space reduces to only 24 rules.

3.4.6 ILASP META-LEVEL REPRESENTATION

ILASP uses a meta-level representation of the learning task that is solved in Clingo. It does so by manipulating the predicates in the program and 'tagging' examples, there are notions of encoding the weights which translate into summing aggregates for each level, and checking for a violating reason can be done with the parity of the single-level penalty. The full encoding is not important for this project, however it is important to note its existence as the grounding of the task is not only dependent on the background knowledge but also the meta-level representation, and there can be repercussions if it is not carefully considered. The complete translation and the definitions of some functions that are used in this report can be found in Law, Russo and Broda (2015a).

META-LEVEL FUNCTIONS

The following two definitions are taken from Law, Russo and Broda (2015a). They are functions that allow the manipulation of predicates in a program, they are used in Chapters 6 and 7 to enable batching and encoding searches.

Definition 3.4.11 (reify). Given a program, Π, a predicate, `pred(·)`, and a term, `term`, $\mathrm{reify}(\Pi, \mathit{pred}, \mathit{term})$ is the program created by replacing all atoms $\alpha \in \mathit{atoms}(\Pi)$ with $\mathit{pred}(a, \mathit{term})$.

---

[10] ILASP actually provides a command line flag `-nc` for this feature, I am expressing it here as a bias constraint for the purpose of example

[11] ILASP also has a flag for a mode declaration `(positive)`, stating that a literal can only appear positively

Definition 3.4.12 (append). Given a program, Π, and an atom α, $\text{append}(\Pi, \alpha) \triangleq \left\{ head(r) \leftarrow body^{+}(r), body^{-}(r), \alpha | r \in \Pi \right\}$.

3.5 GAME THEORY

Within this report references will be made to occasional game theoretic concepts.

3.5.1 GAME TYPES

ZERO-SUM	The reward of a player is the loss of another.
NON-COOPERATIVE	Players are competing against each other.
EXTENSIVE-FORM	Games are played with sequential turns according to some turn function, for example "trailing player goes first" (see the game *Glen More*[12]) or simply "clockwise around the table".

Note. All of the games that are considered in this report are two-person, zero-sum, non-cooperative, extensive-form games.

3.5.2 UTILITY FUNCTIONS

A utility function $u^p : \mathcal{S} \mapsto \mathbb{N}$ is a function from states to natural numbers, with a higher utility indicating that player p prefers this state.

Remark. In zero-sum games there is a single utility function u which is used by all players. Any other player in the game receives the negation of the utility as a reward.

3.5.3 MINIMAX THEOREM

Extensive-form games can be visualised using decision trees, with each node representing a game state and each edge being a legal action from one state to another. The minimax algorithm is a method of selecting an action at a node by assuming what would happen if one's opponent plays optimally, i. e. you choose the move that maximises the utility given the fact your opponent minimises it.

In simple games such as *Nim* and *Tic-Tac-Toe* the tree can be exhaustively searched. Backwards reasoning can be used in order to calculate the exact move to make. Take the example below which describes using the minimax theorem to compute optimal play for the game of *Nim*[13].

Example 3.5.1. Below is a decision tree for the game of *Nim* a game where a player can split any number into two, unless the number is 1 or 2. A player loses if they cannot make a legal move, i. e. left with only 1s and 2s. Figure 3.5 starts the game at the value of 7, leaf nodes are scored with $+1$ for a win and -1 for a loss.

---

[12] `https://boardgamegeek.com/boardgame/66362/glen-more`

[13] `https://wikipedia.com/nim`

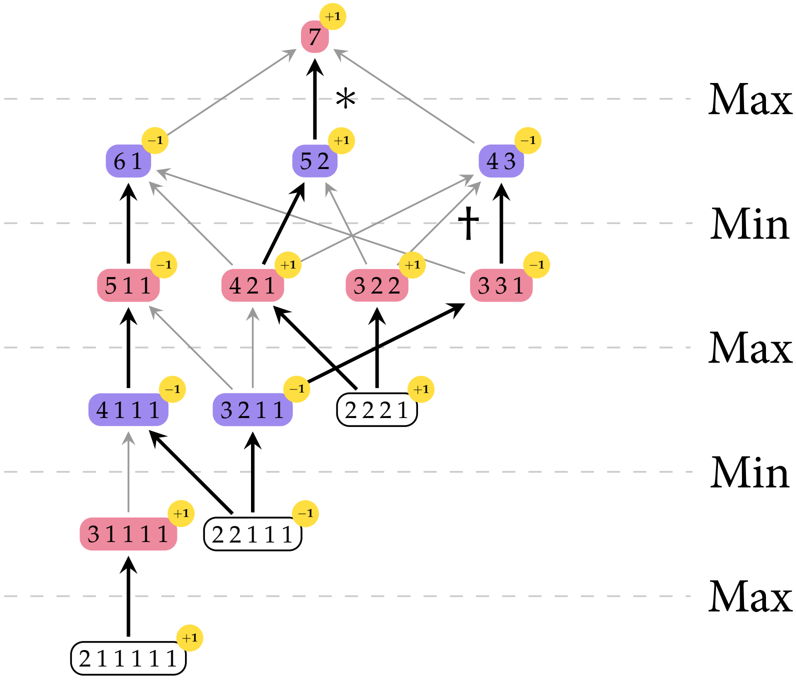


Figure 3.5. Decision tree of *Nim* with backpropagation. Bold arrows denote the backwards propagation of the values in the tree. Coloured nodes represent the player's choice from it's children. White nodes are terminal states. Based on the highlighted arrows the red player should choose the middle ($*$) move as the reward will be $+1$ which is the greatest of the three options $\{+1, -1, -1\}$. (†) shows a min step, the opponent wishes to minimise the proponent's reward and so chooses the -1 action.

However in more complex games, such as *Chess*, *Go* or *Onitama*, the tree is too large and the search would be too computationally intensive. Therefore, the search is only considered up to a certain depth and the states are compared based on the player's utility function.

α-β PRUNING

α-β pruning is an optimisation technique that can be applied to minimax (Heineman, Pollice and Selkow, 2008). It yields exactly the same result, but examines fewer nodes. During the search the algorithm keeps track of two values α and β which denote the highest min score and the lowest max score, respectively. This way if you find something outside of these bounds you need not examine the branch further. This technique is used in the minimax planner in Section 6.1.2. α-β pruning is one of the optimisation techniques applied to Romstad et al. (*Stockfish*), one of the best *Chess* engines.

Take the following example depicted in Figure 3.6. The algorithm starts by setting $\alpha = -\infty$, $\beta = \infty$, and then visits state a and so $\alpha = 6$. Next state b is visited, $3 < 6$ so α stays the same, the maximum of these values, 6, is assigned to state c and $\beta = 6$. The algorithm the traverses the f branch ($\alpha = -\infty$, $\beta = 6$) by visiting d, updating $\alpha = 7$, as $\beta \leq \alpha$ the rest of the branch e is pruned. Using similar logic for the rest of the tree $(*)$ is also pruned.

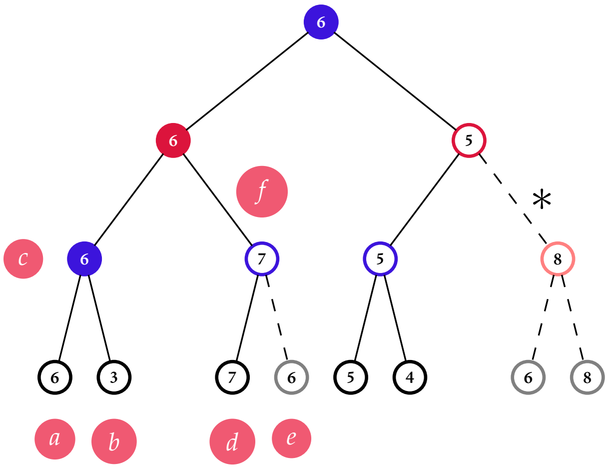


Figure 3.6. α-β pruning tree, dashed lines denote the pruned branches.

HORIZON EFFECT

One common issue with the minimax theorem is the *horizon effect*, named after the leaves of the tree which make up the horizon. Informally, it is the lack of knowing what would happen if you evaluated the tree to the next depth, and how that would affect the result. In practice it means that you may make a move that you thought left you in a strong position but five moves later you realise that if you had thought ahead another move it would have turned out to be a bad move. In this report I ignore this effect, though it is something that could be used in the planning phase (Section 6.1.2) in order to create a more formidable and robust Artificial Intelligence (AI). One method of overcoming the effect is using *quiescent search* is a selective search that looks into tactical play. "Programs with a poor or inadequate quiescence search suffer more from the horizon effect" (Marsland, 1986). In many *Chess* engines this is used applied after some minimax lookahead, commonly it is used to look for checks and captures (creating 'capture trees'). In Chapter 10 we look at defending pieces and exchanges using minimax lookahead. The following chess position studied by Marsland (1986).

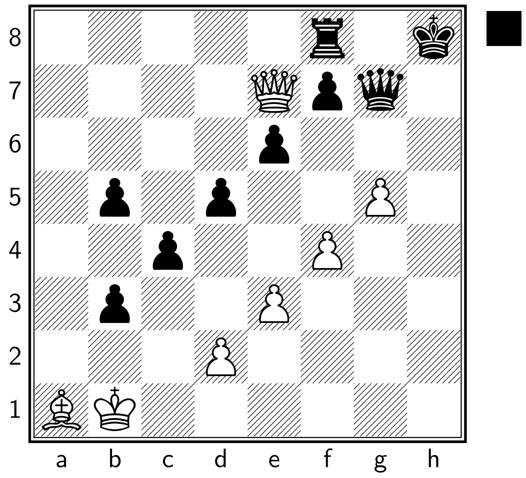


Figure 3.7. Using insufficient quiescent search could result in **1...b2!±Blocking the bishop's attack on the Queen** . However, this just delays the Queen's capture — this is missed by an 8-ply search (**1...b2 2 ♗×b2 c3 3 ♗×c3 d4 4 ♗×d4 e5 5 ♗×e5**). A better variation would be **1...f6**, leading to a draw. FEN: `5r1k/4Qpq1/4p3/1p1p2P1/2p2P2/1p2P3/3P4/BK6 b --`

4 RELATED WORK

Learning strategies in Inductive Logic Programming touches on many areas of computer science such as Logic, Machine Learning, and Game Theory. This diverse background leads to many interesting bodies of work being produced on the topic. Ranging from formal logics to experimental work. I will be looking into areas of Knowledge Representation, use of ASP in planning problems (specifically single-player games), formal logic for describing and verifying properties of strategies, and finally, explainable AIs and 'black box' models.

4.1 KNOWLEDGE REPRESENTATION

There are many considerations to make when choosing how to represent an environment in an ASP logic program. Opting for a more terse representation may mean that you lose some useful information. Conversely, a verbose representation can greatly increase the grounding of the program making the learning task infeasible.

There are several formalisations that can be used to represent games. There exist a series of Action Languages (Gelfond and Lifschitz, 1998) that represent transition systems that are very general (e. g. Fangzhen Lin's suitcase). Some of the languages have translations into ASP (Gebser, Grote and Schaub, 2010; Lee, 2012). However these languages are too general for the simple games. Most games follow similar patterns and can be described using some common features, such as players and actions. These concepts can be expressed in the Action Languages, but there is nothing guiding the representation. This can lead to two games have vastly different representations and therefore varying results when trying to use them in learning tasks.

To avoid this we have chosen to use a language that has been designed for expressing games and is studied and used throughout the General Game Playing (GGP) community to express games. It is simple language that is specific to games, called the Game Description Language (GDL), the specification of which is given in Section 5.2. The basic version covers single player games, later iterations incorporating multi-player games (Love et al., 2006), and incomplete information (Thielscher, 2010). It includes concepts that are specific to games (e. g. `goal` and `role`) which make the logic program more intuitive to read. Translations from GDL to ASP exist (Cerexhe, Sabuncu and Thielscher, 2013) with an initial state, and with timestamped states (defined in terms of the current timestep and actions). A slightly modified version of this translation will be defined in Section 5.2.2 that does away with the time stamps as we are only interested in the current and next state.

4.2 REPRESENTING GAMES IN FORMAL LOGICS

Using the GDL as a basis for further work was completed by Zhang and Thielscher (2015). They formalised the language, allowing them to prove interesting properties of games and extending it to represent strategies of games. The language proposed by Zhang et al. is a modal logic encoding the GDL specification, which intuitively fits the nature of GDL.

The extension of introducing strategies as a subset of the legal moves given some logical "strategy rule" is particularly interesting. Strategy rules can be composed to express priorities using two new logical operators defined in the paper, a prioritised conjunction and a prioritised disjunction.

In their paper, Zhang et al. study various strategies of a simple game, building in complexity. Later, in Chapter 8, I look at learning the presented strategies, comparing ILASP's representation with their own. I also look at applying some of the techniques I describe in Chapter 7 to learn strategies involving their modal operator, which in their paper they are unable to encode strategies for in ASP.

Kaiser (2012) presents a game learning system that learns first order sentences describing the rules (legal moves and outcomes) of several games, including *Connect4*, *Gomoku*[1] and *Tic-Tac-Toe*, by watching videos of the games being played. The system uses a similar structure to the ILP tasks used here, defining relational structures to represent the board state. Kaiser's program successfully learns five games, with the longest learning task taking 906 seconds (excluding video processing). Despite the main objective of the study being to improve upon state-of-the-art visual learning, the use of first order logic and Inductive Logic Programming to describe the games is interesting.

4.3 LEARNING ANSWER SET PROGRAMS

ASP has had success in many constraint and planning problems, however little work has been done using ASP to learn rules, or preferences (as we do in this report). For example (Grasso, Leone and Ricca, 2013) outlines a real world example of a travel website that uses an ASP program to suggest places to go. This program could be improved by learning user preferences of locations and times of year based on previous examples.

4.4 MACHINE LEARNING AND GAMES

Early developments in machine learning systems playing board and video games include Temporal-Difference Backgammon programs (Tesauro, 1995), and Atari video games (Mnih et al., 2013). Artificial intelligence has been prominent in the media recently with *AlphaZero* (Silver et al., 2017a) defeating *Stockfish* in Chess, *Elmo* in Shogi, and its predecessor, *AlphaGo Zero*, (Silver et al., 2017b) in Go. As well as current work looking into playing StarCraft II, a 3-D real-time strategy game (Vinyals et al., 2017).

Many of the examples that have seen public success have been based on deep reinforcement learning, and other statistical methods. The downside of this approach is that they cannot explain the reasoning behind the move. In the case of *AlphaGo Lee*, the version of *AlphaGo* that beat master Go player Lee Sedol, some moves played by *AlphaGo Lee* were beyond the comprehension of expert human players (e. g. game 2 move 37[2]).

4.5 EXPLAINABLE AI

In recent decades we have begun to see an increase in the number of artificially intelligent systems that are equal to or better than human level at complex tasks, e. g. *AlphaZero*, *My-*

---

[1] `https://boardgamegeek.com/boardgame/11929/go-moku`

[2] `https://youtu.be/JNrXgpSEEIE`

cin (Shortliffe, 1977). These complex models are essentially black boxes as far as human interpretation is concerned.

Mycin (Shortliffe, 1977), a medical diagnosis system created in the 1970s by Edward Shortliffe. The program was not rolled out to hospitals due to legal and ethical debates, despite achieving higher accuracy than physicians. The main issue was accountability; no one knew how the algorithm came to its conclusions.

A study by DARPA (a research branch of US defense agency) compared different machine learning methods and how 'explainable' each method is. Figure 4.1 shows a summary of this. Clearly they believe we have some way to go before we can consider our machine learning methods to be fully explainable, having said this ILASP is not an approximation (without noise) of the examples and interpreting the predicates can be done by people with relative ease.

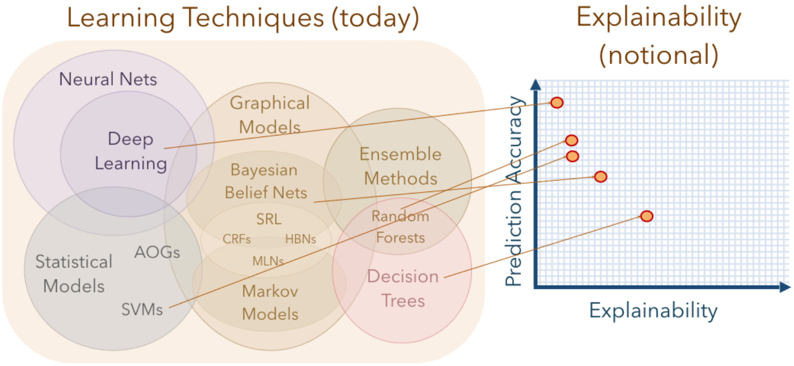


Figure 4.1. Performance vs. Explainability (adapted from Gunning (2016))

Google, in collaboration with Carnegie Mellon University, have looked into explaining how features are perceived by neural networks and visualising neurons to achieve some quite bizarre results (Olah et al., 2018); some examples of the visualisations can be seen in Figure 4.2. Their work also shows key areas of the image that are important in the classification process. For example, a picture of a cottage may be classified as such and the reason is that the neuron for 'house' from Figure 4.2 was highly activated along with a 'thatch' neuron, thereby justifying its decision. Similar work has also been done by others, see Samek, Wiegand and Müller (2017).

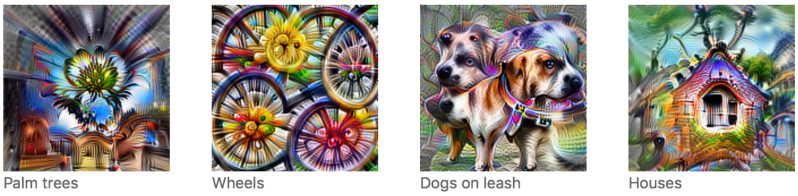


Figure 4.2. Visualisations of neurons from Layer 4c. Figure from Olah et al. (2018)

Part II

IMPLEMENTATION

5 GAME MODEL

All learning tasks require a set of background knowledge, which may be empty.[1] Each context dependent example will also contain some background knowledge specific to the example, i. e. state of the board, at that time. As mentioned in Section 4.1 there is a balance to be struck between preserving information and achieving concision. Section 4.1 also introduced a framework for games; this language is formally introduced here alongside a translation into ASP, the format used by ILASP.

Throughout this section *Onitama* is used as a running example. The other games studied throughout this report (Chapter 2) can be represented similarly, and their full representations can be found in Appendix A.

5.1 INTUITION

In order to represent the state of the board the game's components must be encoded as a series of predicates and function symbols. Outlined below is a list of steps I considered when encoding the game.

1. CREATE TYPES These are simple predicates to define the types of variables later in the program; these are mostly used to ensure that all variables are safe. They could be more rigorous to ensure the rules will make sense if a bad context is given, for example checking a card name is valid, but this is not necessary.

2. CREATE COMPONENTS

BOARD	The board is broken up into a 2-dimensional grid of spaces (or cells).
PAWNS	Pawns are represented as a tuple of their rank and associated player.
CARDS	Cards have a name, a set of moves, and a starting colour associated with them.

3. GAME STATE There are two common ways of representing boards in games: (1) describing what is in each space of the board, (2) describing where each pawn is on the board. In my model of *Onitama* I opted for the latter. This is because there are 25 spaces on the board but only a maximum of 10 pawns that can be on the board, therefore reducing the state space. This is because Clingo computes a subset of the grounding based on the facts in the program. All other spaces are assumed to be empty.

   I also provide what cards players are holding in this state, and the card in play that is not held is the centre card.

4. RULES The predicate `legal(·)` is learnt using examples of legal and illegal moves in different board states. I provide the winning conditions in the background knowledge, though these could also be learnt.

---

[1] Though in all examples presented here this is not the case

5.2 GAME DESCRIPTION LANGUAGE

The Game Description Language (GDL) is a language used in GGP to specify game states and rules. The language is a variant of Datalog with function constants, negation and recursion (Love et al., 2006), though in practice the language Knowledge Interchange Format (KIF) is used, which can very easily be translated into ASP (Section 5.2.2). The advantages of writing the background and examples of the game in this manner are that (a) the system can easily be used in a GGP competition or using past GGP games, (b) as seen in Section 4.2, other work has been done on languages for games derived from GDL, and finally, (c) it is easy to generate the future possible states of the game from the `next` relation which is useful in Chapter 7. The other style that has been used when writing games in ASP is a time based model, allowing answer sets to be paths down the tree. In the end I decided not to use this approach for numerous reasons. The grounding becomes much larger when you increase the number of time steps into the future the game is simulated. The evaluation function evaluates a single state not paths from the current state.

5.2.1 SPECIFICATION

Here I outline the specification as given in Love et al. (2006). KIF statements are written in prefix notation, e. g. `(cell 1 3)` is a function `cell` applied to arguments `1` and `3`. Variables start with question marks, e. g. `(pawn ?rank red)` matches any red pawn. Once again I will use Onitama (Section 2.1) as the running example.

GDL defines several keywords that are used as a common way to represent the states and actions within a game, listed in Table 5.1. The second section of the table lists keywords that are not part of the structure of the game.

KEYWORD	DESCRIPTION
`(role ?r)`	`?r` is a player
`(init ?p)`	`?p` is a predicate true in the initial state of the game
`(true ?p)`	`?p` is a predicate true in the current state of the game
`(next ?p)`	`?p` is a predicate true in the next state of the game, i. e. after an action has been made
`(does ?r ?a)`	`?r` does action `?a`
`(legal ?r ?a)`	`?r` can make action `?a`
`(goal ?r ?v)`	`?r` gets payoff `?v`
`terminal`	This is a terminal state
`(distinct ?a ?b)`	`?a` and `?b` are different
`(not ?p)`	The negation of `?p`
`(<= head body1 body2 ... bodyN)`	Construction of a logical rule

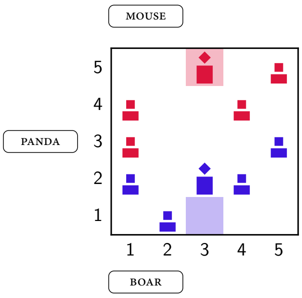
Table 5.1. GDL Keywords

PLAYERS: `role` RELATION

The `role` relation specifies the players in the game, `control` is used to show whose turn it is. In Onitama the roles are `(role red)` and `(role blue)`.

GAME STATE: `true` RELATION

The `true` relation describes what is true before an action is made, i. e. the state of the game. Figure 5.1 shows a game state and its corresponding `true` relation.



```
(true (in_play red (card mouse))
(true (in_play blue (card boar))

(true (location (pawn master red) (cell 5 3))
(true (location (pawn student red) (cell 4 1))
(true (location (pawn student red) (cell 3 1))
(true (location (pawn student red) (cell 4 4))
(true (location (pawn student red) (cell 5 5))
(true (location (pawn master blue) (cell 2 3))
(true (location (pawn student blue) (cell 2 1))
(true (location (pawn student blue) (cell 1 2))
(true (location (pawn student blue) (cell 2 4))
(true (location (pawn student blue) (cell 3 5))

(true (control red))
```

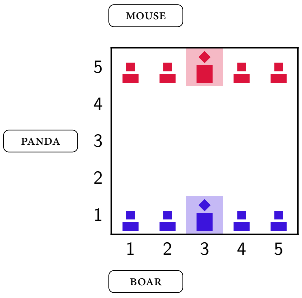
(a) Board state

(b) `true` Relation

Figure 5.1. GDL Current State

INITIAL STATE: `init` RELATION

`init` is similar to `true`, but specific to the initial state of the game. Figure 5.2 shows an example for Onitama.



```
(init (in_play red mouse))
(init (in_play blue boar))

(init (location (pawn master red) (cell 5 3)))
(init (location (pawn student red) (cell 5 1)))
(init (location (pawn student red) (cell 5 2)))
(init (location (pawn student red) (cell 5 4)))
(init (location (pawn student red) (cell 5 5)))
(init (location (pawn master blue) (cell 1 3)))
(init (location (pawn student blue) (cell 1 1)))
(init (location (pawn student blue) (cell 1 2)))
(init (location (pawn student blue) (cell 1 4)))
(init (location (pawn student blue) (cell 1 5)))
```

(a) Initial state

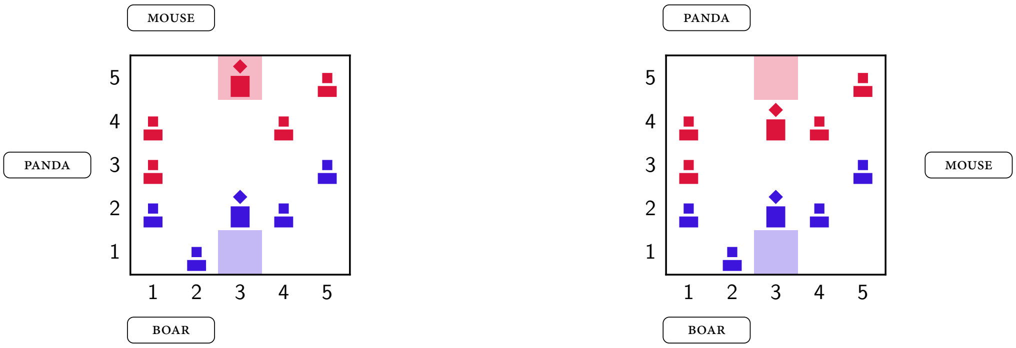
(b) `true` Relation

Figure 5.2. GDL Initial State

GAME STATE UPDATE: `next` RELATION

The `next` relation represents the updates to the game state after an action has been made. This is usually defined as a set of rules, as seen in Figure 5.3c. Figure 5.3 shows an example of this update, where Figure 5.3a shows the current position and the move that is to be made and Figure 5.3b shows the board after the update.

The rules in Figure 5.3c say that if `red` is in control now, `blue` will be in control in the next state and vice versa, i. e. play alternates between the players. The other rule states that a players piece will be at a new location after being moved, (see MOVES: `does` RELATION on page 49). Several rules have been omitted here for simplicity, such as persistence rules and rules regarding the cards.



(**a**) Board state with move (**b**) Next state

Figure 5.3. GDL Next State

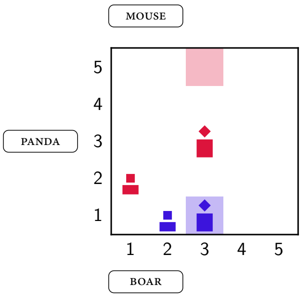
```
(<= (next (control blue)) (true (control red)))
(<= (next (control red)) (true (control blue)))

(<= (next (location (pawn ?rank ?role) ?to))
    (does ?role (move ?from ?to ?card))
    (true location (pawn ?rank ?role) ?from)
)
```

(**c**) `next` Relation

Figure 5.3. GDL Next State

LEGAL MOVES: `legal` RELATION

The `legal` moves relation maps players onto moves that can be made in this state. It is quite common to use the atom `noop` to represent that no action has been made, especially in turn based games. Figure 5.4 shows some examples of the `legal` relation in Onitama.



```
(<= (legal red noop) (true (control blue)))
(<= (legal blue noop) (true (control red)))

(<= (legal ?role (move ?from ?to ?card))
    (true (adj ?from ?to ?card))
    (true (location (pawn ?rank ?role) ?from))
    (not (true (location (pawn ?rank2 ?role) ?to)))
)
```

(a) Board state with legal moves

(b) `legal` Relation

Figure 5.4. GDL Legal and Chosen Moves

MOVES: `does` RELATION

The `does` relation maps players to chosen actions, which will be a subset of the `legal` relation.

Note. Some games allow simultaneous moves by players and some games allow players to make multiple moves, therefore the `does` relation does not necessarily map one player to one action. However, without loss of generality, we can model a player making multiple actions as one combined action.

For example, from Figure 5.4 `red` could choose `(does red (move (cell 2 1) (cell 1 2) mouse))`, and `blue` is forced to choose `(does blue noop)`.

GOAL STATES: `goal` RELATION

```
(<= (goal ?role 100) (win ?role))
(<= (goal ?role 0) (lose ?role))

(<= (lose red) (win blue))
(<= (lose blue) (win red))

; Winning conditions from Section 2.1.1
(<= (win red) (not (true (location (pawn master blue) ?cell))))
(<= (win blue) (not (true (location (pawn master red) ?cell))))
(<= (win red) (true (location (pawn ?rank red) (cell 5 3)))))
(<= (win blue) (true (location (pawn ?rank blue) (cell 1 3)))))
```

Figure 5.5. GDL Goal Relation

TERMINAL STATES: `terminal` RELATION

The `terminal` atom is a nullary predicate that indicates that a state has no actions from this point. This could be due to a winning condition being triggered, a round limit being reached, or lack of available options, for example.

In Onitama the game is over if a player wins, the rules do not state what happens if neither player can move, nor is there a condition where players draw if a certain number of moves have passed without capture (cf. chess). In GDL this would be `(<= terminal (win ?role))`.

5.2.2 TRANSLATION INTO ASP

Here I describe a translation into ASP that maintains the instantaneous nature of GDL (cf. time series models Cerexhe, Sabuncu and Thielscher (2013)). Given the set of GDL rules $\mathcal{G}$, $\llbracket \mathcal{G} \rrbracket = \Pi$, where Π is the equivalent logic program.

$$\llbracket \mathcal{G} \rrbracket = \bigcup_{r \in G} \llbracket r \rrbracket \cup \mathit{generator} \tag{5.1}$$

$$\llbracket \texttt{(<= h b1 ... bn)} \rrbracket = \llbracket \texttt{h} \rrbracket \leftarrow \llbracket \texttt{b1} \rrbracket \; \texttt{...,} \llbracket \texttt{bn} \rrbracket \tag{5.2}$$

$$\llbracket \texttt{(distinct t1 t2)} \rrbracket = \llbracket \texttt{t1} \rrbracket \neq \llbracket \texttt{t2} \rrbracket \tag{5.3}$$

$$\llbracket \texttt{(not a)} \rrbracket = \text{not}\, \llbracket \texttt{a} \rrbracket \tag{5.4}$$

$$\llbracket \texttt{(true (p t1 ... tn))} \rrbracket = p(\llbracket \texttt{t1} \rrbracket, \ldots, \llbracket \texttt{tn} \rrbracket) \tag{5.5}$$

$$\llbracket \texttt{(p t1 ... tn)} \rrbracket = p(\llbracket \texttt{t1} \rrbracket, \ldots, \llbracket \texttt{tn} \rrbracket) \tag{5.6}$$

$$\llbracket \texttt{?v} \rrbracket = V \tag{5.7}$$

$$\llbracket \texttt{a} \rrbracket = a \tag{5.8}$$

where

$$\mathit{generator} = \begin{Bmatrix} 0\{\mathit{does}(\mathit{Role}, A)\}1 \leftarrow \mathit{legal}(\mathit{Role}, A) \\ \leftarrow \mathit{role}(\mathit{Role}), \text{not } \mathit{does}(\mathit{Role}, \_), \text{not } \mathit{terminal} \end{Bmatrix} \tag{5.9}$$

The *generator* set encapsulates the choice of actions and ensures that an action is made at each non-terminal state.

Example 5.2.1. Figure 5.6 shows the translation of Tic-Tac-Toe from GDL into ASP[2].

---

[2] A few rules have been omitted for brevity

```
(role x)
(role o)

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

(<= (next (cell ?m ?n ?role))
    (does ?role (mark ?m ?n))
    (true (cell ?m ?n b)))
(<= (next (cell ?m ?n ?role))
    (true (cell ?m ?n ?role))
    (distinct ?role b))
(<= (next (cell ?m ?n b))
    (does ?role (mark ?j ?k))
    (true (cell ?m ?n b))
    (distinct ?m ?j))
(<= (next (cell ?m ?n b))
    (does ?role (mark ?j ?k))
    (true (cell ?m ?n b))
    (distinct ?n ?k))
(<= (next (control x)) (true (control o)))
(<= (next (control o)) (true (control x)))

(<= open (true (cell ?m ?n b)))

(<= (legal ?role (mark ?x ?y))
    (true (cell ?x ?y b))
    (true (control ?role)))
(<= (legal x noop) (true (control o)))
(<= (legal o noop) (true (control x)))

(<= (goal x 100) (line x))
(<= (goal x 0) (line o))
(<= (goal o 100) (line o))
(<= (goal o 0) (line x))
(<= (goal ?role 50)
    (role ?role) (not open)
    (not (line x)) (not (line o)))

(<= terminal (line ?role))
(<= terminal (not open))
```

(a) Tic-Tac-Toe in KIF adapted from Love et al. (2006)

```
role(x).
role(o).

init(cell(1, 1, b)).
init(cell(1, 2, b)).
init(cell(1, 3, b)).
init(cell(2, 1, b)).
init(cell(2, 2, b)).
init(cell(2, 3, b)).
init(cell(3, 1, b)).
init(cell(3, 2, b)).
init(cell(3, 3, b)).
init(control(x)).

next(cell(M, N, Role) :-
  does(Role, mark(M, N)),
  true(cell(M, N, b)).
next(cell(M, N, Role) :-
  true(cell(M, N, Role)), Role != b.
next(cell(M, N, b) :-
  does(Role, cell(J, K)),
  true(cell(M, N, b), M != J.
next(cell(M, N, b) :-
  does(Role, cell(J, K)),
  true(cell(M, N, b), N != K.
next(control(x)) :- true(control(o)).
next(control(o)) :- true(control(x)).

open :- true(cell(M, N, b)).

0 { does(Role, A) } 1 :- legal(Role, A).
:- role(Role), not does(Role, _),
  not terminal.

legal(Role, mark(X, Y)) :-
  true(cell(X, Y, b)), true(control(Role)).
legal(x, noop) :- true(control(o)).
legal(o, noop) :- true(control(x)).

goal(Role, 100) :- line(Role).
goal(x, 0) :- line(o).
goal(o, 0) :- line(x).

goal(Role, 50) :-
  role(Role), not open,
  not line(x), not line(o).

terminal :- line(Role).
terminal :- not open.
```

(b) Translation of Tic-Tac-Toe into ASP

Figure 5.6. Translation of Tic-Tac-Toe program

5.3 SIMPLIFICATIONS

The representation described in the previous section can become quite large, with many untied[3] variables and a large grounding because of some of the predicates used.

Untied variables are much easier to simplify, clingo and ILASP both have support for anonymous variables (_) which are projected away by the grounder (gringo). This means that, for example, if the following card predicate is used to check ownership of the card the translation would not be needed, allowing us to replace them with anonymous variables.

$$card(Name, DR, DC) \Rightarrow card(Name, \_, \_)$$

By the grounder this is then treated as $card(Name)$ and the arity 3 predicate is now only 1.

Arithmetic expressions can be calculated at ground time by gringo, but when generating mode declarations For example take the following rule for adjacency:

$$\begin{aligned} adj((Row1, Col1), (Row2, Col2)) \leftarrow\ & cell((Row1, Col1)), \\ & cell((Row2, Col2)), \\ & Row2 == Row1 + 1, \\ & Col2 == Col1 + 1. \end{aligned} \tag{5.10}$$

This means that two squares of the form ⧉ are adjacent to each other. However, because the variables cannot be free we must bind the variables by ensuring they are valid cells. On a 5-by-5 board this generates 25 unique cells.

This can be simplified by substituting arithmetic expressions into the head of the equation (where applicable), however the types are still needed.

$$\begin{aligned} adj((Row1, Col1), (Row1 + 1, Col1 + 1)) \leftarrow\ & cell((Row1, Col1)), \\ & cell((Row1 + 1, Col1 + 1)) \end{aligned} \tag{5.11}$$

Clingo is optimised so that these are equivalent when grouding. However when generating the mode declarations in ILASP using the second format is better providing you restrict the maximum length of the body and the number of variables needed. Table 5.2 shows the number of rules generated for the two formats and different parameters passed to ILASP, the $(*)$ indicates that the correct rule does not appear in this set of rules.

Max Variables	Max Weak Constraint Length	Format for rule 5.10	Format for rule 5.11
2	2	1∗	6
4	4	591	2238

Table 5.2. Rules generated for normal/simplified rules with different parameters

---

[3] A word which here means that a variable occurs in the body of a rule only once

6 DIGITISED GAME & PLANNER

In order to collect examples to learn from I created some digital versions of the games that can record different sorts of moves depending on what I was interested in. I created a multi-purpose digital version of each game I was studying: (1) they *collect examples* from recorded games, (2) they utilised learnt strategies to play games in conjunction with a *minimax planner*, (3) and finally they could be used to help the AI learn from it's mistakes with *assistive movement*. The digital games have two running modes and can be configured for human vs. human games, human vs. AI play, and AI vs. AI self-play. When playing a game involving an AI it is possible to run it in tournament-mode or training-mode. Tournament-mode is just like a normal game where the AIs will play the moves they choose based on their current strategy. In training-mode the AI will ask the human whether or not their chosen move is a good one and ask for a suggestion if the human disagrees.

I chose to write the planning library in Haskell as the games can be easily described succinctly using the language's type system. Each game needs a way of getting legal moves, validating if a move is legal, and an update function for moves; I utilised Haskell's Type-Classes in order to make the AI player agnostic to the game.

6.1 PROGRAM FLOW

The main flow of the program is presented in Figure 6.1, the red (🔴), green (🟢) and yellow (🟡) nodes refer to Sections 6.1.1, 6.1.2 and 6.1.3, respectively.

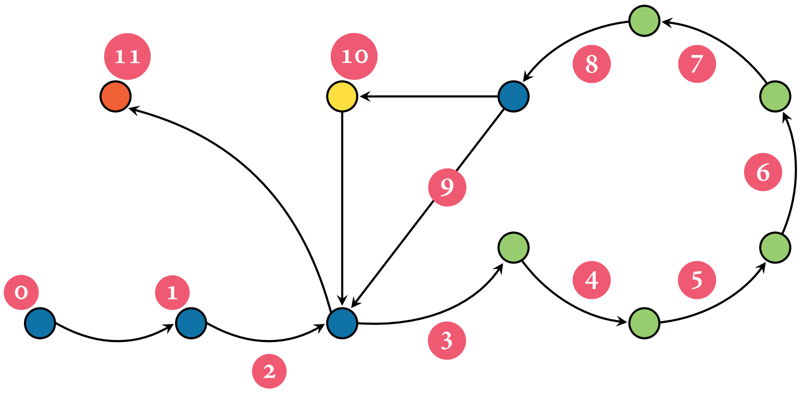


Figure 6.1. User flow through the digital game. Blue states represent intermediate nodes between the sections discussed in this chapter. The red state is the step in which examples are collected, this is at the end of the program after the game has been played. Green states represent the steps made by the minimax planner. Finally, the yellow state is the assistive move step where the user can provide a counter example to the AIs move if needed.

⓪ Select game
① Choose player configurations (human or AI)
② Start Game
③ Human plays move (if applicable)
④ AI generates the game tree to a given depth
⑤ Clingo rules are generated to represent the leaves of the tree, and example is given below (Example 6.1.1)
⑥ Clingo is run in batches of 20,000 to evaluate the scores of the leaves
⑦ The game tree is re-mapped so that leaves now contain scores instead of board states
⑧ The minimax algorithm is run over the tree in order to select the best move
⑨ If in tournament-mode the move is played and it is the next players turn
⑩ If in training-mode the move is presented to the user, if accepted the move is played otherwise the player's chosen move is played and the ordering is saved as a counter example
⑪ If the game is in a terminal state the game is over and (optionally) the winners orderings are saved as described in Section 6.1.1

6.1.1 EXAMPLE COLLECTION

Digital versions of the games can be played by two players and the winner's moves are chosen as examples. We make the assumption that a player has played as well as they could according to their chosen strategy, and that when facing a competent opponent some strategic choices will be made by the winner in order to beat their opposition. The moves they made are then turned into ordering examples in the following way:

1. Create the positive example for their chosen move, m^i_{pref} where i is the turn number, the context is the board state and the inclusions is the single fact `does(·,·)` detailing the move.
2. From the given board state generate all possible alternative moves (i. e. ignoring the move they made).
3. Create the positive examples for alternative moves, $m^i_{alt_j}$ where i is the turn number and j is an index for the move.
4. Create the brave orderings $\langle n, m^i_{pref}, m^i_{alt_j} \rangle \ \forall i \ \forall j$ for some noise weight n.

6.1.2 MINIMAX PLANNER

When calculating which move to make the AI uses Clingo as a scoring mechanism and minimax (Section 3.5.3) as an adversarial search method for looking ahead in the tree of possibilities to better estimate the optimal move.

Example 6.1.1. Given we start in the state pictured in Figure 6.2, and we generate a game tree to depth 2 we could end up in any of the board states pictured in Figure 6.3. In order to score these board states we need Clingo to evaluate the penalty given by the weak constraints that encode the current strategy. Whilst it is possible to do using a single state of the board and getting the score of the single answer set, this method becomes *very* slow when computing the scores of leaves from a depth 4 or more game tree. The bottleneck of this computation is the I/O that must be performed. Therefore, to combat this, the board states can be batched together in the following manner: let B be the background knowledge from Logic Program A.1 excluding the predicates `next(·)` and the action generator, Π_1, Π_2 be two programs representing two board states from Figure 6.3

$$\Pi_1 = \begin{cases} location(pawn(master, red), cell(5,3)) \\ location(pawn(student, red), cell(4,3)) \\ location(pawn(student, red), cell(4,5)) \\ location(pawn(master, blue), cell(1,3)) \\ location(pawn(student, blue), cell(3,2)) \\ location(pawn(student, blue), cell(1,2)) \\ control(red) \\ in\_hand(goose, red) \\ in\_hand(monkey, blue) \end{cases} \quad \Pi_2 = \begin{cases} location(pawn(student, red), cell(5,4)) \\ location(pawn(student, red), cell(4,5)) \\ location(pawn(master, blue), cell(1,3)) \\ location(pawn(student, blue), cell(4,2)) \\ location(pawn(student, blue), cell(3,3)) \\ location(pawn(student, blue), cell(1,2)) \\ control(red) \\ in\_hand(goose, red) \\ in\_hand(monkey, blue) \end{cases}$$

Recall the meta-function append from Section 3.4.6. We use it here to create the program $\Pi = B \cup \text{append}(\Pi_1, option(1)) \cup \text{append}(\Pi_2, option(2))$ which encodes each board state as a different option that Clingo must score.

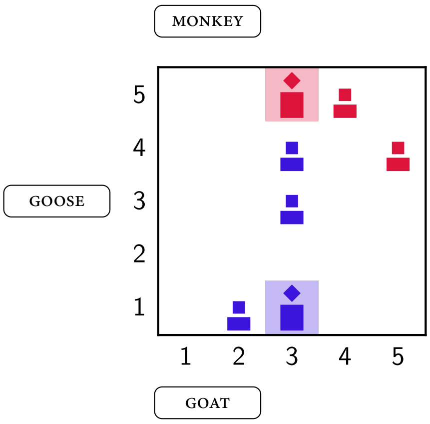
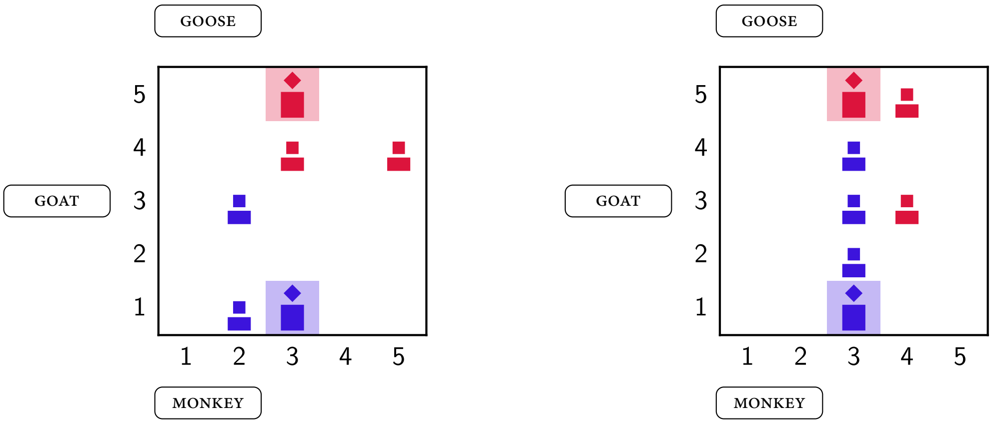


Figure 6.2. Current board state midway through a game

6.1.3 ASSISTIVE MOVEMENT

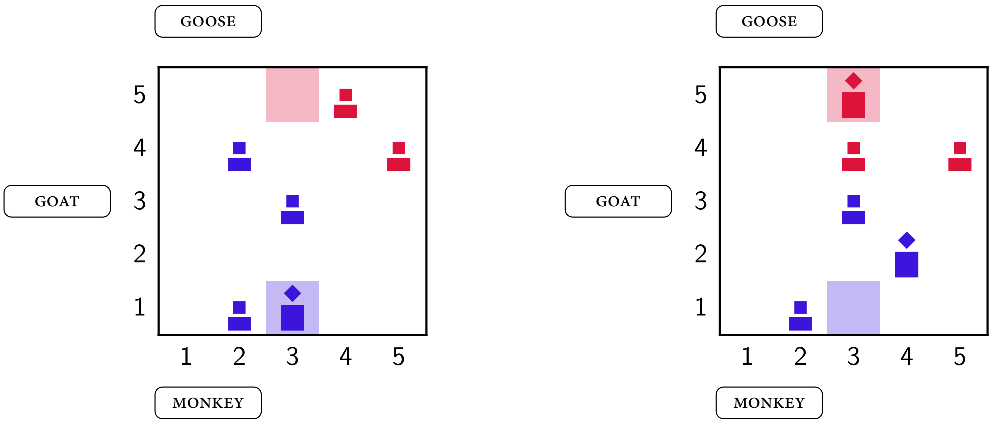
The assistive movement feature was implemented to replicate the style of learning that a parent might take on with their child, or an experienced player with a beginner. When a less experienced player makes a blunder the more experienced player may offer an alternative move and allow them to try to figure out why that move is better. This is what is offered in the training-mode, which when in use, diverts the flow of the program through the yellow (🟡) node in Figure 6.1. After the AI player has selected a move the user is



(a) Figure 6.2 after the moves
♟(5,4) × (4,3) monkey.
♟(3,3) (3,2) goat.

(b) Figure 6.2 after the moves
♟(4,5) (3,4) monkey.
♟(1,2) (2,3) goat.

Figure 6.3. Possible leaf nodes of the game tree pruned to depth 2



(c) Figure 6.2 after the moves
♚(5,3) (4,2) monkey.
♟(4,3) × (4,2) goat.

(d) Figure 6.2 after the moves
♟(5,4) × (4,3) monkey.
♚(1,3) (2,4) goat.

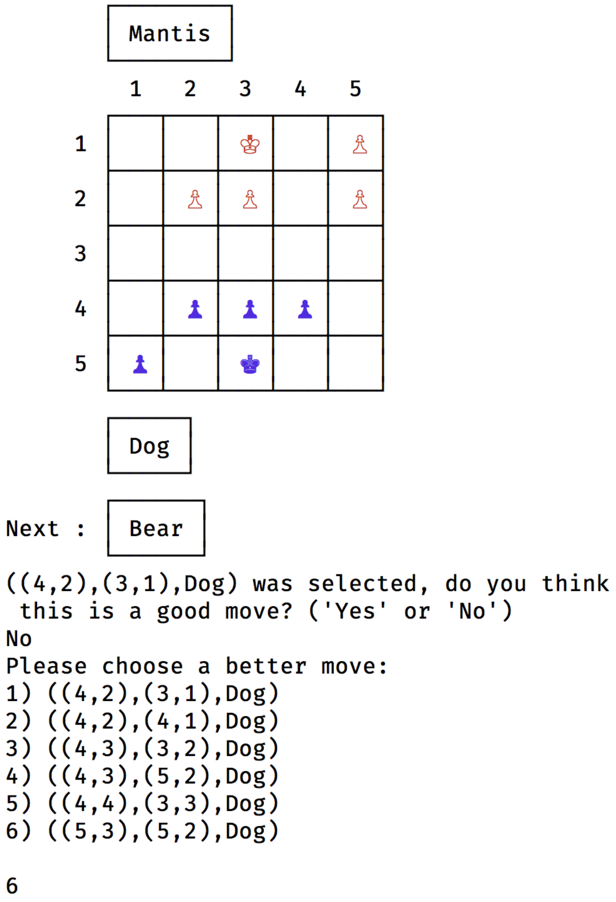
Figure 6.3. Possible leaf nodes of the game tree pruned to depth 2 (cont.)

offered the opportunity to select a correction, Figure 6.4 shows an example of this, the brave ordering created in this example would be:

$$\begin{aligned}
e_1 &= \langle \{ does(blue, move(cell(4,2), cell(3,1), dog)) \}, \emptyset, C \rangle \\
e_2 &= \langle \{ does(blue, move(cell(5,3), cell(5,2), dog)) \}, \emptyset, C \rangle \\
o &= \langle e_1, e_2 \rangle
\end{aligned}$$

where C is the context of the board:

$$C = \left\{ \begin{array}{ll}
location(pawn(student, red), cell(2,2)) & location(pawn(student, blue), cell(5,1)) \\
location(pawn(student, red), cell(2,3)) & location(pawn(student, blue), cell(4,2)) \\
location(pawn(student, red), cell(2,5)) & location(pawn(student, blue), cell(4,3)) \\
location(pawn(student, red), cell(1,5)) & location(pawn(student, blue), cell(4,4)) \\
location(pawn(master, red), cell(1,3)) & location(pawn(master, blue), cell(5,3))
\end{array} \right.$$



```
Mantis

    1   2   3   4   5
1           ♔       ♙
2       ♙   ♙       ♙
3
4       ♟   ♟   ♟
5   ♟       ♚

Dog

Next :  Bear
((4,2),(3,1),Dog) was selected, do you think
 this is a good move? ('Yes' or 'No')
No
Please choose a better move:
1) ((4,2),(3,1),Dog)
2) ((4,2),(4,1),Dog)
3) ((4,3),(3,2),Dog)
4) ((4,3),(5,2),Dog)
5) ((4,4),(3,3),Dog)
6) ((5,3),(5,2),Dog)

6
```

Figure 6.4. Example of a user correcting a move (which directly puts an undefended student in danger) to a more defensive alternative

6.2 EXTENSIBILITY

The digital games and minimax planner have been designed to allow new games to be added easily. The games are implemented as a package within the `Planner.Games.Data` package and a `Main` file responsible for running the game loop and asking for user moves. New AI tactics can be added under the `Planner.Games.AI` package. There are a few additional packages such as `Planner.Games.Data.Tree`, `Planner.Logic`, and `Planner.Minimax` responsible for game trees, translation into (gringo) logic programs, and minimax, respectively.

6.2.1 ADDING NEW GAMES

In addition to *Onitama* I have implemented *Five Field Kono*, and *Cross-Dot*. This can be done by simply describing the games using Haskell's type system and writing a data type that conforms to the `GameState s` type class that is parameterised on a game state, the class uses type families[1] in order to couple the game state type with the types of moves, players, and pieces. The type class is presented below, some methods are necessary for gameplay (e. g. `getBoardArray` and `makeMove`) and some are solely for example collection

---

[1] `https://wiki.haskell.org/GHC/Type_families`

(e. g. `lastMove` and `relevantMoves`). This type class is the used by the game trees, game loop and the AI players to be able to interface with the different game states.

```
class
  (Eq (Player s), Eq (Piece s), Eq (Move s), Show s, Show (Move s), Translate s,
   ↪  Translate (Move s,s), Translate (Player s, Move s), Translate (Player s),
   ↪  Translate (Piece s))
    => GameState s where
  type Move s
  type Player s
  type Piece s
  getBoardArray    :: s -> Board (Piece s)
  getPossibleMoves :: s -> [Move s]
  getResult        :: s -> Maybe (Result (Player s))
  makeMove         :: Move s -> s -> s
  whoseTurn        :: s -> (Player s)
  legalmove        :: Move s -> s -> Bool
  lastMove         :: s -> Maybe (Move s)
  relevantMoves    :: s -> [(Move s, Move s)]
```

Haskell's expressive types mean that lots of parameterised data types could be created that many games can utilise. Below are the types that are used throughout all three games implemented so far.

```
type Location = (Int, Int)
data Position p = Empty | Filled p deriving (Eq, Ord, Show)
type Board p = Array Location (Position p)
data Result p = Draw | Win p
  deriving (Eq, Show)
```

Most abstract strategy games have the notion of a board or grid on which the game is player (e. g. *Chess*, *Go*, *Onitama*, *Tak*[2] etc.). Here the board is represented by an array indexed by a `Location` tuple. Each location might contain a piece, which can be anything at all. In *Five Field Kono* and *Cross-Dot* a piece can be simply denoted by the player. More complex games such as *Chess* and *Onitama* must use a custom piece data type to represent the difference between pieces (e. g. Knights and Rooks or Masters and Students). A game such as *Tak* can even have stacks of different types of pieces, yet here we can still easily represent this, perhaps using `Data.Stack`[3]. The fact that a space *might* contain a piece suggests that the `Maybe`[4] data type could be used. Instead we defined the data type `Position` with two constructors `Empty` and `Filled p`, which is in fact isomorphic to the `Maybe` data type but affording us more clarity when reading the type signatures.

Whilst this framework does not cover all games, it covers most games based around a grid that pieces move on — even the cards in *Onitama* can still be represented and used. This could be abstracted further by allowing the game to represent the indexing into the board array or the result, but the extra generality covers a set of games that are not yet being looked at by the learning tasks presented in this report.

---

[2] https://boardgamegeek.com/boardgame/197405/tak

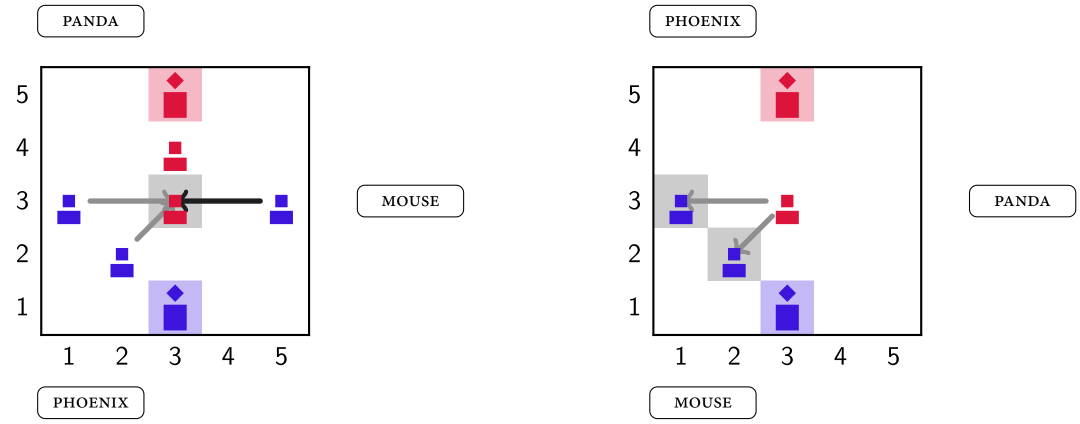
[3] http://hackage.haskell.org/package/Stack-0.3.2/docs/Data-Stack.html

[4] http://hackage.haskell.org/package/base-4.11.1.0/docs/Data-Maybe.html

7 LEARNING PREFERENCES FROM GAME TREES

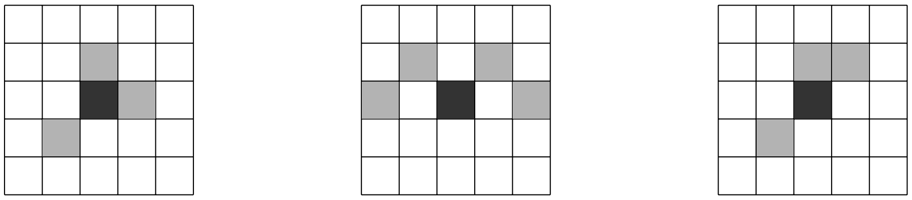
7.1 MOTIVATION

In many games a player can gain a strategic advantage by considering possible moves in the future, consider chess where Grand Masters are often calculating games 5–10 steps into the future in their head at any time. In some games it is very easy to think ahead due to, say, only a few moving pieces on the board. Whereas others can quickly branch and it is difficult to consider all possibilities, so you may choose just a few likely options to consider. Take, for example, this situation in Onitama:



(a) Chosen move in dark grey and alternative move in light grey.

(b) Blue to move following the exchange ♙$(5,3) \times (3,3)$, ♙$(3,4) \times (3,3)$



(c) MOUSE **(d)** PHOENIX **(e)** PANDA

Figure 7.1. Possibility of an exchange

Example 7.1.1. Figure 7.1a depicts blue's turn with the PHOENIX card. If blue's strategy was to "capture a piece if possible, otherwise move randomly" then blue would pick one of the three moves shown. Suppose blue chooses ♙$(5,3) \times (3,3)$, one possible series of (unfortunate) events that could unfold for blue is shown in Figure 7.1b. From this position blue cannot capture red, and has few pawns, thus few options. Furthermore, no matter what move blue makes red is able to take a piece weakening blue's position, this tactic is known as a *fork*. However, if we consider the move again but looking a few steps into the future we can see that even though that pawn is in danger, it is possible to pick a move that avoids this scenario and puts the blue player into a good position.

Note. When reasoning like this using minimax you can suffer from the *horizon effect* (Section 3.5.3), imagine that this was in fact the state we got to after look several moves into the future and was the best option. If we had looked *just one more level* further we would have seen a different result. Thus, looking further down the tree is not guaranteed to always give better results.

Looking into this effect is not in the scope of this project, but is an interesting area for further work (see Section 11.2.1).

7.2 INDUCTIVE LEARNING PROGRAMS WITH DEEP ORDERINGS

In this section we define a new ordering type that extends Context Dependent Ordering Example (CDOE) to take this tree structure of games into account when learning. In all the learning tasks we have seen up to this point all the examples have been independent from each other. In order to reason in the future it is important to consider possible outcomes at future states, and so we may wish to look for certain preferences within groups of examples that represent states in the future. In order to define these groups we must create a relation between a subset of the examples. The new orderings use minimax trees to work out the best strategy for the player. We can then say that a player prefers a move to another iff it is the best action to take given the minimax tree from this state. These trees will define the relationship between certain examples.

Note. We make the assumption that both players are rational and using the same utility.

A *deep* ordering takes two states and looks down the game tree to a certain depth. In order to illustrate this consider the following trees in Figures 7.2, 7.3 and 7.4. The root node is the game board in Figure 7.1. Dashed lines represent a choice, where only one node is needed as part of the requirement, solid lines mean that all nodes are considered.

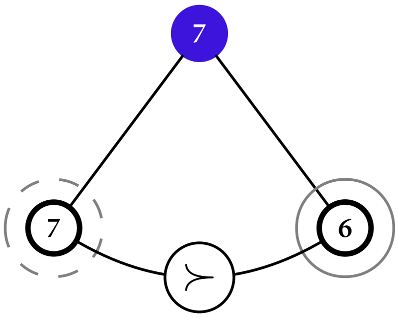


Figure 7.2. Play is considered to depth 0

Suppose Alan is playing a game and chooses the *left* action, assuming Alan is rational and playing optimally we can deduce that the action *left* was a better choice than action *right*. Figure 7.2 illustrates this choice, with the numbers in the nodes representing the utility Alan gains for choosing that action. Here we see that the utility of *left* is greater than that of *right*.

For Alan to have known what the utility of *left* was he would have had to considered possible moves his opponent, Betty, could have made. Alan can then reason about which action to take assuming Betty is trying to make him lose and she is also rational and playing optimally. For the *left* action to be the optimal choice it must be the case that no matter what Betty chooses to do it cannot be worse for him than one of the options she would have, had he chosen the *right* action. Figure 7.4 illustrates what this might look like. We see that the *left* action has expanded into two actions, *l-left* and *l-right*, from which Betty can choose between. Further, we can observe that *all* options reachable from *left* are better (i. e. have a higher utility) than the single option, *r-only*, reachable from *right*.

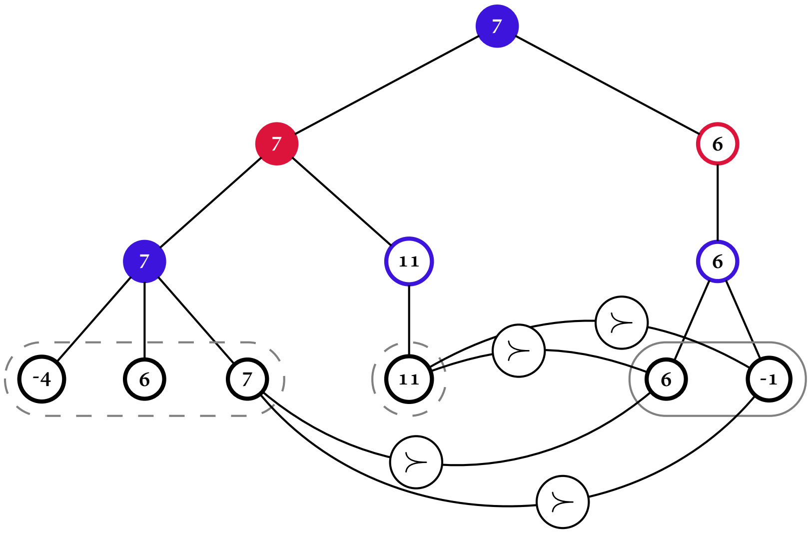


Figure 7.4. Play is considered to depth 2

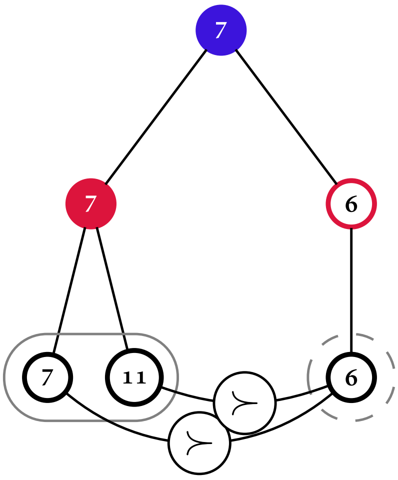


Figure 7.3. Play is considered to depth 1

However, notice that this can now be broken into two sub games, one for the game after Alan selects *left* and one after Alan selects *right*. Betty, in both cases, can do as Alan did and consider possible moves that Alan can respond with. This leads us onto Figure 7.4.

In Figure 7.4 we once again unfold the tree down one more level. This now shows the choices Alan has. Given Alan chose action *left*, and assuming Betty selected the *l-left*, Alan can choose between *l-l-left*, *l-l-middle*, and *l-l-right*. As Alan is rational we know that Alan will choose the option that is better than any of the options he would be faced with given the action Betty could have chosen if he had originally selected *right*. In other words, one of *l-l-left*, *l-l-middle*, and *l-l-right*, must be better than both *r-o-left* and *r-o-right* (the actions highlighted with a solid line in Figure 7.4). However, this must also hold if Betty had chosen action *l-right*! Therefore, *l-r-only* must also be better than both *r-o-left* and *r-o-right*.

We can clearly from the diagram that this holds, and so the action *left* that Alan originally selected was the optimal move (considering up to a depth of two). It is also apparent how this recursive structure can be generalised to an arbitrary depth, and arbitrary branching factors.

Formally, we represent the game trees, $\mathcal{T}$, as a tuple $(e, \lhd)$ where $e \in \mathcal{S}$ is the root of the tree, a state of the game board, and $\lhd : \mathcal{S} \times \mathcal{S}$ the child relation ($c \lhd p$ reads "c is a child of p") which is a partial ordering over $\mathcal{S}$, where $\mathcal{S}$ is the set of all states of the game board. Let $u \colon \mathcal{S} \mapsto \mathbb{N}$ be the utility function of the game.

Definition 7.2.1 (Minimax Phases). The minimax algorithm is broken up into a *max_phase* and a *min_phase*, where *max_phase*, *min_phase* are functions with signature $\mathcal{T} \mapsto \mathbb{N}$.

$$\mathit{max\_phase}((e, \lhd), d) \triangleq \begin{cases} u(e) & \text{if } d = 0 \\ \max \left\{ \mathit{min\_phase}(e', d-1) \mid e' \lhd e \right\} & \text{otherwise} \end{cases}$$

$$\mathit{min\_phase}((e, \lhd), d) \triangleq \begin{cases} u(e) & \text{if } d = 0 \\ \min \left\{ \mathit{max\_phase}(e', d-1) \mid e' \lhd e \right\} & \text{otherwise} \end{cases}$$

A variation of the minimax algorithm that selects between just two children of a node is defined as:

$$\begin{aligned} &\mathit{minimax}\colon \mathcal{T} \times \mathcal{T} \times \mathbb{N} \mapsto \mathcal{T} \\ &\mathit{minimax}((e_1, \lhd), (e_2, \lhd), d) \triangleq \arg\max \begin{Bmatrix} \mathit{min\_phase}((e_1, \lhd), d), \\ \mathit{min\_phase}((e_2, \lhd), d) \end{Bmatrix}, \end{aligned}$$

where $d \geq 0$.

Definition 7.2.2 (Explanation Condition). The following *explanation condition* is a predicate that takes a utility function, two trees and a depth and will be true if there are paths in the left tree that explain why the left branch is preferred to the right branch. This is done by alternating quantifiers at each depth, following the intuition given previously.

$$\begin{aligned} &\mathit{explanation}\colon (\mathcal{S} \mapsto \mathbb{N}) \times \mathcal{T} \times \mathcal{T} \times N \mapsto \mathit{Bool} \\ &\mathit{explanation}(u, (e_1, \lhd), (e_2, \lhd), d) \\ &\quad \triangleq (\forall e_1^1 \lhd e_1)(\exists e_1^2 \lhd e_1^1)(\forall e_1^3 \lhd e_1^2) \cdots (\forall e_1^{d-1} \lhd e_1^{d-2})(\exists e_1^d \lhd e_1^{d-1}) \\ &\quad\quad (\exists e_2^1 \lhd e_2)(\forall e_2^2 \lhd e_2^1)(\exists e_2^3 \lhd e_2^2) \cdots (\exists e_2^{d-1} \lhd e_2^{d-2})(\forall e_2^d \lhd e_2^{d-1}) \\ &\quad\quad u(e_1^d) > u(e_2^d) \end{aligned}$$

Proposition 1. *Given two trees* $(e_1, \lhd)$, $(e_2, \lhd)$, *a depth, d, and a utility function, u,*

$$\mathit{min\_phase}((e_1, \lhd), d) > \mathit{min\_phase}((e_2, \lhd), d) \iff \mathit{explanation}(u, (e_1, \lhd), (e_2, \lhd), d)$$

Proof.

Case 1 (Base Case).

Case 1.1 ($d = 0$). To show:

$$\mathit{min\_phase}((e_1, \lhd), 0) > \mathit{min\_phase}((e_2, \lhd), 0) \iff \mathit{explanation}(u, (e_1, \lhd), (e_2, \lhd), 0)$$

From the definition of *min_phase* we get that $u(e_1) > u(e_2)$, which is the definition of $\mathit{explanation}(u, (e_1, \lhd), (e_2, \lhd), 0)$, as when $d = 0$ there are no quantifiers.

Case 1.2 ($d = 1$). To show:

$$\mathit{min\_phase}((e_1, \lhd), 1) > \mathit{min\_phase}((e_2, \lhd), 1)$$

$$\iff (\forall e_1^1 \lhd e_1)(\exists e_2^1 \lhd e_2) u(e_1^d) > u(e_2^d)$$

$$min\_phase((e_1, \lhd), 1) > min\_phase((e_2, \lhd), 1) \tag{7.1}$$

$$\iff \min_{e_1^1 \lhd e_1} \left(max\_phase(e_1^1, 0) \right) > \min_{e_2^1 \lhd e_2} \left(max\_phase(e_2^1, 0) \right)$$

$$\iff (\forall e_1^1 \lhd e_1)(\exists e_2^1 \lhd e_2)\ max\_phase(e_1^1, 0) > max\_phase(e_2^1, 0)$$

This follows from being able to select any child of e_1 as it is true for the minimum and the fact that one can select the child of e_2 that yields the minimum

$$\iff (\forall e_1^1 \lhd e_1)(\exists e_2^1 \lhd e_2)\ u(e_1^1) > u(e_2^1) \quad \text{(from definition } max\_phase\text{)} \tag{7.2}$$

Case 2 (Inductive Step, $d \geq 2$). For any trees $(f_1, \lhd)$ and $(f_2, \lhd)$ let the inductive hypothesis be:

$$IH \triangleq min\_phase((f_1, \lhd), d-2) > min\_phase((f_2, \lhd), d-2)$$

$$\iff (\forall f_1^1 \lhd f_1)(\exists f_1^2 \lhd f_1^1)(\forall f_1^3 \lhd f_1^2) \cdots (\forall f_1^{d-3} \lhd f_1^{d-4})(\exists f_1^{d-2} \lhd f_1^{d-3})$$

$$(\exists f_2^1 \lhd f_2)(\forall f_2^2 \lhd f_2^1)(\exists f_2^3 \lhd f_2^2) \cdots (\exists f_2^{d-3} \lhd f_2^{d-4})(\forall f_2^{d-2} \lhd f_2^{d-3})$$

$$u(f_1^{d-2}) > u(f_2^{d-2})$$

To show:

$$min\_phase((e_1, \lhd), d) > min\_phase((e_2, \lhd), d)$$

$$\iff (\forall e_1^1 \lhd e_1)(\exists e_1^2 \lhd e_1^1)(\forall e_1^3 \lhd e_1^2) \cdots (\forall e_1^{d-1} \lhd e_1^{d-2})(\exists e_1^{d} \lhd e_1^{d-1})$$

$$(\exists e_2^1 \lhd e_2)(\forall e_2^2 \lhd e_2^1)(\exists e_2^3 \lhd e_2^2) \cdots (\exists e_2^{d-1} \lhd e_2^{d-2})(\forall e_2^{d} \lhd e_2^{d-1})$$

$$u(e_1^d) > u(e_2^d)$$

$$min\_phase((e_1, \lhd), d) > min\_phase((e_2, \lhd), d) \tag{7.3}$$

$$\iff \min_{e_1^1 \lhd e_1} \left(max\_phase(e_1^1, d-1) \right) > \min_{e_2^1 \lhd e_2} \left(max\_phase(e_2^1, d-1) \right) \tag{7.4}$$

$$\iff (\forall e_1^1 \lhd e_1)(\exists e_2^1 \lhd e_2) \tag{7.5}$$

$$max\_phase(e_1^1, d-1) > max\_phase(e_2^1, d-1)$$

$$\iff (\forall e_1^1 \lhd e_1^1)(\exists e_2^1 \lhd e_2)$$

$$\max_{e_1^2 \lhd e_1^1} \left(min\_phase(e_1^1, d-2) \right) > \max_{e_2^2 \lhd e_2^1} \left(min\_phase(e_2^2, d-2) \right) \tag{7.6}$$

$$\begin{aligned}\iff & (\forall e_1^1 \lhd e_1)(\exists e_2^1 \lhd e_2)(\exists e_1^2 \lhd e_1^1)(\forall e_2^2 \lhd e_2^1) \\ & \qquad min\_phase(e_1^2, d-2) > min\_phase(e_2^2, d-2)\end{aligned} \tag{7.7}$$

Using the inductive hypothesis, where $(f_1, \lhd) = (e_1^2, \lhd)$ and $(f_2, \lhd) = (e_2^2, \lhd)$, yields:

$$\begin{aligned}\iff & (\forall e_1^1 \lhd e_1)(\exists e_2^1 \lhd e_2)(\exists e_1^2 \lhd e_1^1)(\forall e_2^2 \lhd e_2^1) \\ & (\forall e_1^3 \lhd e_1^2)(\exists e_1^4 \lhd e_1^3)(\forall e_1^5 \lhd e_1^4)\cdots(\forall e_1^{d-1} \lhd e_1^{d-2})(\exists e_1^d \lhd e_1^{d-1}) \\ & (\exists e_2^3 \lhd e_2^2)(\forall e_2^3 \lhd e_2^2)(\exists e_2^4 \lhd e_2^3)\cdots(\exists e_2^{d-1} \lhd e_2^{d-2})(\forall e_2^d \lhd e_2^{d-1}) \\ & u(e_1^d) > u(e_2^d)\end{aligned} \tag{7.8}$$

$$\begin{aligned}\iff & (\forall e_1^1 \lhd e_1)(\exists e_1^2 \lhd e_1^1)(\forall e_1^3 \lhd e_1^2)\cdots(\forall e_1^{d-1} \lhd e_1^{d-2})(\exists e_1^d \lhd e_1^{d-1}) \\ & (\exists e_2^1 \lhd e_2)(\forall e_2^2 \lhd e_2^1)(\exists e_2^3 \lhd e_2^2)\cdots(\exists e_2^{d-1} \lhd e_2^{d-2})(\forall e_2^d \lhd e_2^{d-1}) \\ & u(e_1^d) > u(e_2^d)\end{aligned} \tag{7.9}$$

Equation 7.9 follows from the fact you can always choose the child that produces the maximum/minimum and $(\forall a)(\forall b)\ P(a,b) \equiv (\forall b)(\forall a)\ P(a,b)$ and $(\exists a)(\exists b)\ P(a,b) \equiv (\exists b)(\exists a)\ P(a,b)$.

□

Now, let the states be Context Dependent Partial Interpretations and we can define a new ILP_{LOAS}^{deep} task that uses the explanation condition to select a subset of the leaves of a game tree to bravely respect, these preferences *explain* why at the root of the tree there is a preference, according to the minimax theorem. In order to define this new task we must first define a new ordering type that incorporates *explanation*. To do so we must first alter the definition of *explanation* – in a manner that preserves Proposition 1 – by replacing the comparison of utilities $u(e_1^d) > u(e_2^d)$ with $B \cup H$ bravely respects $\langle e_1^d, e_2^d \rangle$, where $B \cup H$ are given. This can be done as there is a mapping from the penalties given by the weak constraints to the natural numbers.

Definition 7.2.3 (Deep Context Dependent Ordering Example)**.** This definition extends Definition 3.4.9. A deep CDOE, o, is a tuple $\langle e_1, e_2, d, \lhd \rangle$ of two trees, $(e_1, \lhd)$ and $(e_2, \lhd)$, where e_1, e_2 are CDPIS, $\lhd$ is a child relation over the CDPIS, and a depth d. An ASP program Π *deeply* respects o iff $explanation(\Pi, (e_1, \lhd), (e_2, \lhd), d)$.

Definition 7.2.4 (Deep, Context Dependent LOAS task)**.** This definition extends Definition 3.4.10. A deep, context dependent Learning from Ordered Answer Sets task is a tuple $T = \langle B, S_M, E, O, \lhd \rangle$ where B, S_M, E are as before in Definition 3.4.10. $O = \langle O^b, O^c, O^d \rangle$ where O^d is the set of deep orderings over $\mathcal{S} \subseteq E^+$, $\mathcal{S}$ is the set of all board states.[1] $\lhd$ is a partial ordering over $\mathcal{S}$ denoting the child relation. A hypothesis H is an inductive solution of T, $H \in ILP^{deep}_{\text{LOAS}}$ iff:

(i) $H' \in ILP^{context}_{\text{LAS}}(\langle B, S^{\text{LAS}}_{M}, E^{+}, E^{-}\rangle)$, where H' is the subset of H with no weak constraints, $M^{\text{LAS}} = \langle M_h, M_b \rangle$.

(ii) $\forall o \in O^{b}$, such that $B \cup H$ bravely respects o

(iii) $\forall o \in O^{c}$, such that $B \cup H$ cautiously respects o

(iv) $\forall o \in O^{d}$, such that $B \cup H$ deeply respects o.

7.3 IMPLEMENTATION

The implementation of the new ILP^{deep}_{LOAS} task is detailed below. Firstly we describe an experimental feature of ILASP 3.2 (meta-program injection), This feature allows certain examples to be activated, through the custom definition of the `example_active(·)` predicate. From here we present the injected meta-program we use to encode the explanation condition, and prove that this is equivalent to the ILP^{deep}_{LOAS} task. We also present a method of generating the child relation, and board states from the GDL specification.

7.3.1 ILASP WITH META-PROGRAM INJECTION

In order to be able to modify the meta-level representation we must use the meta-program injection feature of ILASP. The following definition of the feature is from (Law, Russo and Broda, 2018):

Definition 7.3.1 (Meta-Program Injection). Let $T = \langle B, S_M, \langle E^{+}, E^{-}, O^{b}, O^{c}\rangle\rangle$ be an $ILP^{context}_{\text{LOAS}}$ task and $\mathcal{Q}$ be an ASP program. Given any interpretation I of $\mathcal{Q}$, we T_I denotes the task $\langle B, S_M, \langle E^{+}_{I}, E^{-}_{I}, O^{b}_{I}, O^{c}_{I}\rangle\rangle$, where:

$$\begin{aligned}
E^{+}_{I} &= \{e \in E^{+} \mid example\_active(e_{id}) \in I\}\\
E^{-}_{I} &= \{e \in E^{-} \mid example\_active(e_{id}) \in I\}\\
O^{b}_{I} &= \{o \in O^{b} \mid example\_active(e_{id}) \in I\}\\
O^{c}_{I} &= \{o \in O^{c} \mid example\_active(e_{id}) \in I\}
\end{aligned}$$

A hypothesis $H \subseteq S_M$ is said to be an *inductive solution* of T with respect to the injection of $\mathcal{Q}$ iff $\exists I \in \mathcal{AS}(\mathcal{Q})$ such that $H \in ILP^{context}_{\text{LOAS}}(T_I)$.

7.4 TRANSLATION FROM DEEP, CONTEXT DEPENDENT LOAS TASK TO CONTEXT DEPENDENT LOAS TASK WITH META-PROGRAM INJECTION

For any ILP^{deep}_{LOAS} task, $T = \langle B, S_M, E, O, \lhd\rangle$, we translate T to $T' = \langle B, S_M, \langle E^{+}, E^{-}, O^{b}_{inject}, O^{c}\rangle\rangle$ the $ILP^{context}_{\text{LOAS}}$ with $\mathcal{Q}$ as follows:

---

[1] When presenting examples only the board states that are used will be shown

E^+, E^- and O^c are left unchanged. O^b_{leaves} is the set of all possible combinations of leaves at depth d on the left branch and leaves at depth d on the right branch

$$O^b_{leaves} = \bigcup_{\langle e_1, e_2, d, \lhd \rangle \in O^d} \left\{ \langle e^d_1, e^d_2 \rangle \mid \forall e^d_1 \lhd^d e_1, \forall e^d_2 \lhd^d e_2 \right\}$$

$$\begin{aligned} O^b_{inject} &= O^b \cup O^b_{leaves} \\ \mathcal{Q} &= meta(E^+) \cup meta(E^-) \cup meta(O^b) \cup meta(O^b_{leaves}) \\ &\quad \cup meta(O^c) \cup meta(O^d) \cup meta(\lhd) \end{aligned}$$

$\lhd^d$ is the d^{th} power of $\lhd$, defined inductively by

$$\begin{aligned} \lhd^0 &= \{(e, e) \mid e \in \mathcal{S}\} \\ \lhd^1 &= \lhd \\ \lhd^{i+1} &= \lhd \circ \lhd^i \qquad \text{(for } i > 0, \circ \text{ is functional composition)} \end{aligned}$$

and the *meta* function is as follows:

$$meta(E^+) = \{example\_active(e_{id}) \mid e_{id} \in E^+\} \tag{7.10}$$

$$meta(E^-) = \{example\_active(e_{id}) \mid e_{id} \in E^-\} \tag{7.11}$$

$$meta(O^b) = \left\{example\_active(o_{id}) \mid o_{id} \in O^b\right\} \tag{7.12}$$

$$meta(O^c) = \{example\_active(o_{id}) \mid o_{id} \in O^c\} \tag{7.13}$$

$$meta(O^b_{leaves}) = \left\{ ord(e^1_{id}, e^2_{id}, o_{id}) \,\middle|\, o = \langle e^1, e^2 \rangle \in O^b_{leaves} \right\} \tag{7.14}$$

$$meta(O^d) = \left\{ \begin{matrix} root(e_1, chosen) \\ root(e_2, other) \end{matrix} \,\middle|\, o \in O^d \right\} \tag{7.15}$$

$$meta(\lhd) = \{child(a, b) \mid b \lhd a\} \tag{7.16}$$

Equations 7.10–7.13 add facts to the meta-program ensuring that the positive, negative, brave and cautious orderings must be covered. Equation 7.14 adds a set of facts denoting the possible orderings of the leaves at depth d. Equation 7.15 adds the root trees to the meta-program. Finally, we add a fixed program responsible for searching through all the possible brave orderings, O^d_{leaves}, in order to find the set

$$\bigcup_{\langle e_1, e_2, d, \lhd \rangle \in O^d} \left\{ \langle e^d_1, e^d_2 \rangle \,\middle|\, explanation(B \cup H, (e_1, \lhd), (e_2, \lhd), d) \right\}$$

for a hypothesis, H, that will be learnt by the task. This can be achieved using a choice rule for selection over the children of a branch such that $\exists e^{i+1} \lhd e^i$, and a normal rule for enforcing $\forall e^{i+1} \lhd e^i$.

$$explanation(EX\_ID, forall) \leftarrow root(EX\_ID, chosen) \tag{7.17}$$

$$explanation(EX\_ID, exists) \leftarrow root(EX\_ID, other) \tag{7.18}$$

$$\begin{aligned}1\{\mathit{explanation}(\mathit{Child},\mathit{forall}) \mid \mathit{child}(\mathit{Parent},\mathit{Child})\}1& \\ \leftarrow \mathit{explanation}(\mathit{Parent},\mathit{exists}),& \\ \mathit{child}(\mathit{Parent},\_)&\end{aligned} \tag{7.19}$$

$$\begin{aligned}\mathit{explanation}(\mathit{Child},\mathit{exists}) \leftarrow\ & \mathit{child}(\mathit{Parent},\mathit{Child}), \\ & \mathit{explanation}(\mathit{Parent},\mathit{forall})\end{aligned} \tag{7.20}$$

$$\begin{aligned}\mathit{example\_active}(\mathit{ORD\_ID}) \leftarrow\ & \mathit{ord}(\mathit{EX\_ID\_1},\mathit{EX\_ID\_2},\mathit{ORD\_ID}), \\ & \mathit{explanation}(\mathit{EX\_ID\_1},\_), \\ & \mathit{explanation}(\mathit{EX\_ID\_2},\_).\end{aligned} \tag{7.21}$$

Proposition 2. *Given the task $T = \langle B, S_M, E, O, \lhd\rangle$, the meta-program $\mathcal{Q}$ from the translation of T into a meta-program injection task, and a hypothesis H, $\exists I \in \mathcal{AS}(\mathcal{Q})$ such that*

$$\forall o \in O^b_{leaves}(I),\ B \cup H \text{ bravely respects } o \iff \forall o \in O^d,\ B \cup H \text{ deeply respects } o$$

where $O^b_{leaves}(I) = \left\{o \in O^b_{leaves} \mid \mathit{example\_active}(o_{id}) \in I\right\}$

Proof. Let $T = \langle B, S_M, E, O, \lhd\rangle$, $\mathcal{Q}$ be the translation of T into a meta-program injection task, and hypothesis H.

$$(\forall o \in O^d) B \cup H \text{ deeply respects } o \tag{7.22}$$

$$\iff (\forall o \in O^d)(\forall o' \in O^b_{leaves}(I,o)) B \cup H \text{ bravely respects } o \tag{7.23}$$

$$\iff I \in \mathcal{AS}(\mathcal{Q})\ \forall \mathit{example\_active}(o_{id}) \in I\ .\ H \text{ covers } o \tag{7.24}$$

$$\iff \forall o \in O^b_{leaves}(I) B \cup H \text{ bravely respects } o \tag{7.25}$$

Equation 7.24 comes from the fact that Rules 7.20 and 7.19 clearly generate all the children and exactly one child, respectively. All the terms needed to satisfy these predicates are given as facts in $\mathcal{Q}$ (see $\mathit{meta}(O^d)$, $\mathit{meta}(O^b_{leaves})$, $\mathit{meta}(\lhd)$). We know that $B \cup H$ bravely respects this ordering from the explanation condition. *Note.* The upper bound of 1 is not required, it just limits the number of examples that are generated, the important bound is the lower bound, stating that *there must be at least one.* □

Theorem 1. *Given the task $T = \langle B, S_M, E, O, \lhd\rangle$ and the task T' with $\mathcal{Q}$ created from the translation of T into a meta-program injection task then for all $H \subseteq S_M$,*

$$H \in ILP^{deep}_{\text{LOAS}}(T) \iff H \in ILP^{context}_{\text{LOAS}}(T') \text{ with } \mathcal{Q}$$

Proof. In order for $H \in ILP^{deep}_{\text{LOAS}}(T)$ it must satisfy the following (from Definition 7.2.4):

(i) $H' \in ILP^{context}_{\text{LAS}}(\langle B, S^{\text{LAS}}_M, E^+, E^-\rangle)$, where H' is the subset of H with no weak constraints, $M^{\text{LAS}} = \langle M_h, M_b\rangle$.

(ii) $\forall o \in O^b$, $B \cup H$ bravely respects o

(iii) $\forall o \in O^c$, $B \cup H$ cautiously respects o

(iv) $\forall o \in O^d$, $B \cup H$ deeply respects o.

In order for $H \in ILP^{context}_{\text{LOAS}}(T')$ with $\mathcal{Q}$ the following must hold: $\exists I \in \mathcal{AS}(\mathcal{Q})$ such that $H \in ILP^{context}_{\text{LOAS}}(T'_I)$, and for $H \in ILP^{context}_{\text{LOAS}}(T'_I)$ the following must be satisfied (from Definition 7.3.1, Definition 3.4.7):

(v) $H' \in ILP^{context}_{\text{LAS}}(\langle B, S_M^{\text{LAS}}, E_I^{+\prime}, E_I^{-\prime} \rangle)$, where H' is the subset of H with no weak constraints, $M^{\text{LAS}} = \langle M_h, M_b \rangle$.

(vi) $\forall o \in O_I^{b\prime}$, $B \cup H$ bravely respects o

(vii) $\forall o \in O_I^{c\prime}$, $B \cup H$ cautiously respects o

It remains to show that (i) $\Leftrightarrow$ (v), (iii) $\Leftrightarrow$ (vii), and (ii) and (iv) $\Leftrightarrow$ (vi).

1. (i) $\Leftrightarrow$ (v) B and S_M are left unchanged by the translation. From $meta(E^+)$, $\forall e \in E^+$ the fact $example\_active(e_{id})$ is in $\mathcal{Q}$ therefore it is also in I therefore $\forall e \,.\, e \in E^+ \Leftrightarrow e \in E_I^{+\prime}$. A similar argument can be made for E^-.
2. (iii) $\Leftrightarrow$ (vii) A similar argument as above shows that $\forall o \,.\, o \in O^c \Leftrightarrow O_I^{c\prime}$.
3. (ii) and (iv) $\Leftrightarrow$ (vi) $O_I^{b\prime}$ is split, by definition, into $O_I^b \cup O^b_{leaves}(I)$. Again, using a similar argument to 1. and 2., it is sufficient to show that

$$\forall o \in O^b_{leaves}(I), B \cup H \text{ bravely respects } o \iff \forall o \in O^d, B \cup H \text{ deeply respects } o$$

where $O^b_{leaves}(I) = \{o \in O^b_{leaves} \mid example\_active(o_{id}) \in I\}$. This follows directly from Proposition 2.

□

7.5 AUTOMATICALLY GENERATING THE GAME TREES

As the depth and number of deep orderings increases the number of examples and orderings that are needed grow rapidly. Fortunately, the structure of the GDL-based logic programs can be leveraged in order to generate the additional positive examples in the tree, and the child relation, $\lhd$, automatically. Each root example can be combined with its context and the background and using Clingo the children can be created using the methodology outlined below.

We define two mutually recursive functions (algorithm 7.1 and algorithm 7.2). One which for each answer set of of an example will generate the branches and collect the child relation and all leaf examples from the branches. The other will take an answer set an extract the next state (i. e. take the `next(·)` and make that the current state), and then aggregate the results of making the branches from each child.

The *Branch Generation* algorithm is the entry point of the generation, a root node is passed in along with the background knowledge and a depth to generate. If $d = 0$ then the entry node is the child, which is already in the examples, and so empty sets are returned. Otherwise, the *Child Generation* algorithm is called for each answer set of $B \cup C$, this is done by calling out to Clingo, and the results are then aggregated.

Algorithm 7.1: Branch Generation

```
Input: Background Knowledge: $B$
Input: Positive Example: $\langle e, C\rangle \in \mathcal{S}$
Input: Depth: $d \geq 0$
Result: A set of $\langle e, C\rangle$ for each leaf at depth $d$, and the child relation, $\lhd$
begin
    if $d = 0$ then
        return $\emptyset, \emptyset$
    else
        $(decendents, E^{+}_{child}) \longleftarrow$
          $unzip\,\{\text{ChildGen}(B, e_{id}, d-1, A) \mid A \in \mathcal{AS}(B \cup C)\}$
        return $\bigcup decendents,\ \bigcup E^{+}_{child}$
    end
end
```

Algorithm 7.2: Children Generation

```
Input: Background Knowledge: $B$
Input: Positive Example ID: $e_{id}$
Input: Depth: $d \geq 0$
Input: Answer Set: $A$
Result: Tuple of the child relation, $\lhd$, and the child examples, $E^{+}_{child}$
begin
    $state \longleftarrow \{f \mid next(f) \in A\}$
    $E^{+}_{child} \longleftarrow \{\langle\{move\}, \emptyset, state\rangle \mid move \in \text{legal}(state)\}$
    $\lhd \longleftarrow \left\{(e'_{id}, e_{id}) \mid e' \in E^{+}_{child}\right\}$
    if $d = 0$ then
        return $\lhd$, $E^{+}_{child}$
    else
        $(decendents, E^{+}_{leaves}) \longleftarrow unzip\left\{\text{BranchGen}(B, e', d) \mid e' \in E^{+}_{child}\right\}$
        return $\bigcup_{\lhd' \in decendents \cup \lhd} \lhd',\ \bigcup E^{+}_{leaves}$
    end
end
```

The *Child Generation* algorithm takes as input the background knowledge, the ID of the current example and an Answer Set of the example and the background knowledge. The depth, d, is also passed in, and is the value we reduce on in order to terminate. Firstly the next state is created from the answer set, and the child examples and child relation are generated from this state. Here we use the *legal* which simply returns the set of legal moves for the game, in practice this can be done by, again, calling out to Clingo.

Example 7.5.1. Take the following tree from Chapter 8, it depicts a game tree for the CrossDot game (Section 2.3). The left branch denotes the chosen path by the player, and the right branch denotes another possible move. The strategy here is that not letting your opponent win is a good idea. If, however, you were playing by the strategy "move into an isolated box or next to your own marker" (a dominant strategy for the first player) then the right branch would be preferred.

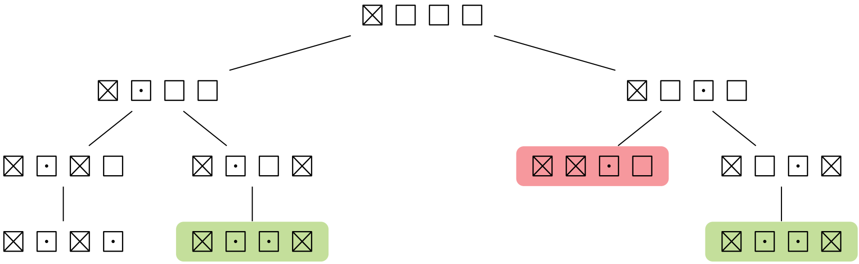


Figure 7.5. Cross-Dot *defence* game tree generated from $\langle(\cdot, \boxtimes\ \square\ \square\ \square, a_2), (\cdot, \boxtimes\ \square\ \square\ \square, a_3), 2\rangle$, the ordering example, with a depth of 2. Green and red represent a winning and losing state for $\cdot$, respectively.

The background knowledge, B, for this task can be found in Logic Program A.3. The mode declaration is $M = \langle\emptyset, \emptyset, \{goal(\mathrm{v}_{role}, 100)\}, \{1, -1\}, 1\rangle$. The only positive examples, E^+, given are:

$$e_1 = \langle\emptyset, \emptyset, C_1\rangle$$

$$e_2 = \langle\emptyset, \emptyset, C_2\rangle$$

$$C_1 = \begin{cases} box(1,x). \quad box(2,b). \quad box(3,b). \quad box(4,b). \\ control(o). \\ \leftarrow \text{not } does(o, mark(2)). \end{cases}$$

$$C_2 = \begin{cases} box(1,x). \quad box(2,b). \quad box(3,b). \quad box(4,b). \\ control(o). \\ \leftarrow \text{not } does(o, mark(3)). \end{cases}$$

The only deep ordering example given is $\langle e_1, e_2, 2, \lhd\rangle$. No other examples are given. The full task is $T = \langle B, S_M, E, O, \lhd\rangle$.

Next we generate the additional children examples and the child relation $\lhd$.

$$e_{1,1,1} = \langle\emptyset, \emptyset, C_3\rangle \qquad e_{1,2,1} = \langle\emptyset, \emptyset, C_4\rangle \qquad e_{2,1,1} = \langle\emptyset, \emptyset, C_5\rangle \qquad e_{2,2,1} = \langle\emptyset, \emptyset, C_6\rangle$$

$$C_3 = \begin{cases} box(1,x). \quad box(2,o). \quad box(3,b). \quad box(4,x). \\ control(o). \\ \leftarrow \text{not } does(o, mark(4)). \end{cases}$$

$$C_4 = \begin{cases} box(1,x). \quad box(2,o). \quad box(3,x). \quad box(4,b). \\ control(o). \\ \leftarrow \text{not } does(o, mark(3)). \end{cases}$$

$$C_5 = \begin{cases} box(1,x). \quad box(2,x). \quad box(3,o). \quad box(4,b). \\ control(o). \\ \leftarrow \text{not } does(o, noop). \end{cases}$$

$$C_6 = \begin{cases} box(1,x). \quad box(2,b). \quad box(3,o). \quad box(4,x). \\ control(o). \\ \leftarrow \text{not } does(o, mark(2)). \end{cases}$$

$$\lhd = \begin{cases} (e_1, e_{1,1}), & (e_1, e_{1,2}), \\ (e_2, e_{2,1}), & (e_2, e_{2,2}), \\ (e_{1,1}, e_{1,1,1}), & (e_{1,2}, e_{1,2,1}), \\ (e_{2,1}, e_{2,1,1}), & (e_{2,2}, e_{2,2,1}), \end{cases}$$

All e are added to E^+ and $\lhd$ is as defined above. Next we translate T into T' the meta-program injection task. $T' = \langle B, S_M, \langle E^+, E^-, O^b_{inject}, O^c \rangle \rangle$ with $\mathcal{Q}$. E^+, E^-, and O^c are unchanged, as per the translation.

$$O^b_{inject} = O^b \cup \begin{Bmatrix} \langle e_{1,1,1}, e_{2,1,1} \rangle, & \langle e_{1,1,1}, e_{2,2,1}, \rangle \\ \langle e_{1,2,1}, e_{2,1,1} \rangle, & \langle e_{1,2,1}, e_{2,2,1} \rangle \end{Bmatrix}$$

$$\mathcal{Q} = \begin{cases} root(e_1, chosen) \qquad root(e_2, other) \\ \\ ord(e_{1,1,1}, e_{2,1,1}, o_1) \quad ord(e_{1,1,1}, e_{2,2,1}, o_2) \\ ord(e_{1,2,1}, e_{2,1,1}, o_3) \quad ord(e_{1,2,1}, e_{2,2,1}, o_4) \\ \\ example\_active(e_1) \qquad example\_active(e_2) \\ example\_active(e_{1,1,1}) \quad example\_active(e_{1,2,1}) \\ example\_active(e_{2,1,1}) \quad example\_active(e_{2,2,1}) \\ \\ explanation(E, forall) \leftarrow root(E, chosen) \\ explanation(E, exists) \leftarrow root(E, other) \\ \\ 1\{explanation(C, forall) \mid child(P,C)\}1 \leftarrow child(P,\_), explanation(P, forall) \\ explanation(C, exists) \leftarrow child(P,C), explanation(P, forall) \\ \\ example\_active(O) \leftarrow ord(E1, E2, O), explanation(E1,\_), explanation(E2,\_) \end{cases}$$

The solution of the task is H iff $\exists I \in \mathcal{AS}(\mathcal{Q})$ such that $H \in ILP^{context}_{\text{LOAS}}(T'_I)$. H is the hypothesis below:

$$\leftsquigarrow goal(P, 100), control(P).[-1@1, P]$$

and $I \in \mathcal{AS}(\mathcal{Q})$ contains exactly the following $example\_active(o_{id})$ predicates (and other predicates, e. g. *root* etc.): $I_{example\_active} = \{example\_active(o_1), example\_active(o_2)\}$

8 CASE STUDY: CROSS-DOT GAME

In this chapter I focus on an alternative game, which I have taken from the literature. The game I will look at has also been studied in Zhang and Thielscher (2015). The Cross-Dot Game, another name for a m–k game, where players aim to get k boxes in a row marked with their player symbol (from a possible m boxes), the full description and rules of the game can be found in Section 2.3. An example of a game state may look as follows: $\boxtimes \square \boxdot \square \square$.

As discussed in Section 4.2, Zhang and Thielscher (2015) present an extension to Game Description Language which includes a modal operator ($\bigcirc$), the ability to represent strategies in their language (not just the game rules), some preference operators for the strategies ($\vartriangle$, $\triangledown$) and a translation of a subset of their language into ASP.

Zhang et al. use the following definition of a strategy. A strategy of player i is $S(\varphi) \subseteq W \times A^i$ where W is the set of game states, A^i is the set of actions player i can take and φ is a strategy rule that is true in all ω such that $(\omega, a) \in S(\varphi)$. A strategy S is only valid if $S(\varphi) \neq \emptyset$.

The notion of prioritised conjunction ($\vartriangle$) and prioritised disjunction ($\triangledown$) were introduced by Zhang and Thielscher in order to combine smaller strategy rules into more complex strategies. The semantics of $r_1 \vartriangle \cdots \vartriangle r_n$ are: apply as many strategy rules from the left as possible whilst still maintaining a valid strategy for this state. The semantics of $r_1 \triangledown \cdots \triangledown r_n$ are: try r_1 and if that does not work try r_2 and so on.

In their paper Zhang et al. present strategies written in their modal language, they then compare them and show interesting properties of each. The experiments in this chapter take examples of games that have been played using the strategies and learn an equivalent strategy in ASP. Experiments 8.1–8.5 use context-dependent LOAS tasks and Experiment 8.6 onwards use our deep, context-dependent LOAS tasks from Chapter 7.

8.1 THE GAME

The game state is denoted by a tuple of current player and current state $(p,\ \square \square \cdots \square)$. For instance, $(\times,\ \boxtimes \boxdot \square \square)$ where $\times$ is to move and each player has had one turn playing in the first two boxes. Moves, or state-action pairs, are represented as a tuple of state and action $(p,\ \square \square \cdots \square, a_b)$ where a_b represents marking box b.

Example 8.1.1. Taking the example game state above and the action a_3, we get the move $(\times,\ \boxtimes \boxdot \square \square, a_3)$. When this move is made the following state change occurs: $(\times,\ \boxtimes \boxdot \square \square) \rightarrow^{a_3} (\cdot,\ \boxtimes \boxdot \boxtimes \square)$. If we preferred this move to $(\times,\ \boxtimes \boxdot \square \square, a_4)$ we can express this as the CDOE (Definition 3.4.9): $\langle (\times,\ \boxtimes \boxdot \square \square, a_3), (\times,\ \boxtimes \boxdot \square \square, a_4) \rangle$. This notation is used throughout the chapter.

8.1.1 REPRESENTATION

To represent the game I have used a modified version of the ASP presented in the paper. This is to eliminate certain duplication, and due to some restrictions imposed by the ASP allowed in ILASP, e. g. conditions. The representation I have used follows from the GDL specification in Section 5.2. `next` means the predicate is true after the move. `does` means a move is taken. `legal` means a move is possible given `true`. The representation used in the paper has multiple rules for some predicates that deal with boxes that can be on either side; which I overcome by introducing `adj(·,·)` meaning two boxes are adjacent. Additionally, I introduce `longest_chain(·,·)` to denote a player's longest chain to avoid creating multiple instances of the program for different ks. The full logic program used as background knowledge for the game can be found in Logic Program A.3.

8.2 LEARNING STRATEGIES

In order to demonstrate the capabilities of weak constraints I use ILASP 3 to learn the strategies presented and compare the actions chosen with the actions selected using the strategies from Zhang and Thielscher (2015). The strategies in the paper can be divided into three categories: simple, prioritised and forward-thinking. These strategy rules are shown in Table 8.1. In all the examples in the following section partial interpretations are written in a simplified format using the aforementioned state-action representation.

Note. Throughout this chapter I use E_O^+ to denote the positive examples drawn from the ordering examples (i. e. O^b, O^c and O^d).

SIMPLE	PRIORITISED	FORWARD-THINKING
fill_next	*combined*	*defence*
fill_isolated	*fill_leftmost*	*cautious*
fill_any	*thoughtful*	*fill_o_next*
		passive_defence

Table 8.1. Categorised strategy rules from Zhang and Thielscher (2015)

8.2.1 SIMPLE STRATEGIES

Simple strategies can be learnt using only a single priority level for the weak constraints.

Experiment 8.1 (Fill Next). The *fill_next* strategy means if possible, mark a box next to one of your own. Given the Context Dependent LOAS task (Definition 3.4.10) $T_{next} = \langle B, S_M, E_O^+, \emptyset, O^b, \emptyset \rangle$ where

B is the background knowledge from Logic Program A.3,

$$M = \langle \emptyset, \emptyset, \{adj(v,v), does(c, mark(v)), box(v,c)\}, \{-1\}, 1 \rangle$$

$$O^b = \begin{cases} \langle (\times, \boxtimes\ \square\ \square\ \square\ \square\ \boxdot, a_2), (\times, \boxtimes\ \square\ \square\ \square\ \square\ \boxdot, a_3) \rangle \\ \langle (\times, \boxdot\ \square\ \boxtimes\ \square\ \square\ \square, a_2), (\times, \boxdot\ \square\ \boxtimes\ \square\ \square\ \square, a_5) \rangle \\ \langle (\times, \boxdot\ \square\ \boxtimes\ \square\ \square\ \square, a_2), (\times, \boxdot\ \square\ \boxtimes\ \square\ \square\ \square, a_6) \rangle \\ \langle (\times, \boxdot\ \square\ \boxtimes\ \square\ \square\ \square, a_4), (\times, \boxdot\ \square\ \boxtimes\ \square\ \square\ \square, a_5) \rangle \\ \langle (\times, \square\ \boxdot\ \square\ \square\ \boxtimes\ \square, a_6), (\times, \square\ \boxdot\ \square\ \square\ \boxtimes\ \square, a_3) \rangle \end{cases}$$

The ordering examples shown here are taken from a few games against a player (·) using a random strategy. Each example is from the × player's point of view.

ILASP was able to learning the following hypothesis[1]:

$$\leftsquigarrow does(x, mark(Box1)), adj(Box1, Box2), box(Box2, x).[-1@1, Box1, Box2]$$

In English this hypothesis means you[2] gain a reward of 1 if you mark a box next to a box you have already marked.

Experiment 8.2 (Fill isolated). The *fill_next* strategy has a flaw, if you play in the first box your opponent can easily block you off. To remedy this you may wish to play in one of the centre cells as you are guaranteed to win on your next turn (assuming $k = 2$). The *fill_isolated* strategy means if possible, mark a box with no mark either side of the box. Given the learning task $T_{iso} = \langle B, S_M, E_O^+, \emptyset, O^b, \emptyset \rangle$ where

B, M are the same as in T_{next},

$$O^b = \begin{cases} \langle (\times, \square\,\square\,\square\,\square\,\square\,\square, a_2), (\times, \square\,\square\,\square\,\square\,\square\,\square, a_1) \rangle \\ \langle (\times, \square\,\square\,\square\,\square\,\square\,\square, a_2), (\times, \square\,\square\,\square\,\square\,\square\,\square, a_6) \rangle \\ \langle (\times, \square\,\square\,\square\,\square\,\square\,\square, a_3), (\times, \square\,\square\,\square\,\square\,\square\,\square, a_6) \rangle \\ \langle (\times, \boxtimes\,\boxdot\,\square\,\square\,\square\,\square, a_4), (\times, \boxtimes\,\boxdot\,\square\,\square\,\square\,\square, a_3) \rangle \end{cases}$$

The examples here show what you might play at the beginning of the game with this strategy, and what you might do later in the game if you made a mistake earlier on.

ILASP was able to learning the following hypothesis[1]:

$$\leftsquigarrow does(x, mark(Box1)), adj(Box1, Box2), true(box(Box2, b)).[-1@1, Box1, Box2]$$

In English this hypothesis means you[2] try to maximise the number of blank boxes either side of you.

Remark. *fill_any* expresses a lack of preference over the actions and so does not need anything to be learnt.

8.2.2 COMBINED STRATEGIES

Combined strategies use the prioritised operators $\triangle$ and $\triangledown$ to compose basic notions into something more complicated.

Experiment 8.3 (Combined). The *combined* is the composite strategy *fill_next* $\triangledown$ *fill_isolated* $\triangledown$ *fill_any*, meaning you first try to fill a box next to your own, then fill an isolated box and if there are no moves that conform to this strategy you fill any box you can. Given the learning task $T_{comb} = \langle B, S_M, E_O^+, \emptyset, O^b, \emptyset \rangle$ where

B is the same as in T_{next},

$$M = \langle \emptyset, \emptyset, \{adj(v, v), does(c, mark(v)), true(box(v, c))\}, \{-1, box\}, 2 \rangle$$

$$O^b = O^b_{next} \cup O^b_{iso}$$

---

[1] Variable names have been replaced for clarity

[2] Here we assume you are playing as ×

Note. Here we are able to use the union of the examples used for the simple strategies (O^b_{next}, O^b_{iso}) as they encode the preference of *fill_next* over *fill_isolated*. Specifically, the example $\langle(\times, \boxtimes \square \square \square \square \boxdot, a_2), (\times, \boxtimes \square \square \square \square \boxdot, a_3)\rangle$, and there are no counter examples to this preference. Though this is not always the case.

ILASP 3 was able to learning the following hypothesis[1]:

$$\leftsquigarrow does(x, mark(Box1)), adj(Box1, Box2), true(box(Box2, x)).[-1@2, Box1, Box2]$$
$$\leftsquigarrow does(x, mark(Box1)), adj(Box1, Box2), true(box(Box2, b)).[-1@1, Box1, Box2]$$

In English this hypothesis means you[2] try to go next to your own piece first, and if this is not possible minimise the number of dots next to you. I believe this is equivalent to the other strategy in this game.

Experiment 8.4 (Fill Leftmost). The *fill_leftmost* is the composite strategy $c_m \vartriangle \cdots \vartriangle c_1$ where $c_i = \bigvee_{j=1}^{i} does(a_j)$ such that $1 \leq i \leq m$ (e. g. if $does(a_2)$ is an option at the current state ω and $does(a_1)$ is not then $(\omega, a_2) \in S(c_m \vartriangle \cdots \vartriangle c_1)$ but $(\omega, a_{-2}) \notin S(c_m \vartriangle \cdots \vartriangle c_1)$ where -2 is everything bar 2).

Remark. This representation of this strategy is rather unintuitive, we will see that in ILASP this is far simpler and more self explanatory.

Given the learning task $T_{left} = \langle B, S_M, E^+_O, \emptyset, O^b, \emptyset\rangle$ where

B, M are the same as in T_{comb},
O^b $= \{\langle(\times, \boxtimes \boxdot \square \square \boxdot \boxtimes, a_3), (\times, \boxtimes \boxdot \square \square \boxdot \boxtimes, a_4)\rangle\}$

ILASP 3 was able to learning the following hypothesis[1]:

$$\leftsquigarrow does(x, mark(Box1)).[Box1@1, Box1]$$

In English, this hypothesis means you gain a penalty proportional to the position of the box, i. e. the further left the lower the penalty.

Experiment 8.5 (Thoughtful). The *thoughtful* strategy means the same as the combined strategy but in all cases pick the left most option when faced with a choice. Formally this is, *combined* $\vartriangle$ *fill_leftmost*. Given the learning task $T_{thoughtful} = \langle B, S_M, E^+_O, \emptyset, O^b, \emptyset\rangle$ where

$$
\begin{aligned}
B &= B_{comb} \cup \{\mathit{leftmost}(B, 7 - B) \leftarrow \mathit{box}(B)\}, \\
M &= \langle M^h, M^b, M^o, \{-1, \mathit{box}, \mathit{score}\}, 2\rangle \\
M^h &= \{\mathit{fill\_next}(\mathrm{v}_{box}, \mathrm{v}_{box}), \mathit{fill\_isolated}(\mathrm{v}_{box}, \mathrm{v}_{box})\} \\
M^b &= \{\mathit{adj}(\mathrm{v}_{box}, \mathrm{v}_{box}), \mathit{does}(\mathrm{c}_{role}, \mathit{mark}(\mathrm{v}_{box})), \mathit{true}(\mathit{box}(\mathrm{v}_{box}, \mathrm{c}_{state})\} \\
M^o &= \{\mathit{fill\_next}(\mathrm{v}_{box}, \mathrm{v}_{box}), \mathit{fill\_isolated}(\mathrm{v}_{box}, \mathrm{v}_{box}), \mathit{leftmost}(\mathrm{v}_{box}, \mathrm{v}_{box})\}
\end{aligned}
$$

$$
O^b = \begin{cases}
\langle(\times, \square\ \square\ \square\ \square\ \square\ \square, a_2), (\times, \square\ \square\ \square\ \square\ \square\ \square, a_3)\rangle \\
\langle(\times, \square\ \square\ \square\ \square\ \square\ \square, a_2), (\times, \square\ \square\ \square\ \square\ \square\ \square, a_1)\rangle \\
\langle(\times, \square\ \boxtimes\ \boxdot\ \square\ \square\ \square, a_1), (\times, \square\ \boxtimes\ \boxdot\ \square\ \square\ \square, a_5)\rangle \\
\langle(\times, \boxdot\ \boxtimes\ \square\ \square\ \square\ \square, a_3), (\times, \boxdot\ \boxtimes\ \square\ \square\ \square\ \square, a_4)\rangle \\
\langle(\times, \square\ \boxtimes\ \square\ \boxdot\ \square\ \square, a_1), (\times, \square\ \boxtimes\ \square\ \boxdot\ \square\ \square, a_3)\rangle \\
\\
\langle(\times, \square\ \square\ \boxtimes\ \boxdot\ \square\ \square, a_2), (\times, \square\ \square\ \boxtimes\ \boxdot\ \square\ \square, a_1)\rangle \\
\langle(\times, \square\ \square\ \boxtimes\ \boxdot\ \square\ \square, a_2), (\times, \square\ \square\ \boxtimes\ \boxdot\ \square\ \square, a_6)\rangle \\
\langle(\times, \square\ \boxdot\ \boxtimes\ \square\ \square\ \square, a_4), (\times, \square\ \boxdot\ \boxtimes\ \square\ \square\ \square, a_1)\rangle \\
\langle(\times, \square\ \boxdot\ \boxtimes\ \square\ \square\ \square, a_4), (\times, \square\ \boxdot\ \boxtimes\ \square\ \square\ \square, a_5)\rangle \\
\langle(\times, \boxdot\ \square\ \boxtimes\ \square\ \square\ \square, a_2), (\times, \boxdot\ \square\ \boxtimes\ \square\ \square\ \square, a_6)\rangle
\end{cases}
$$

O^b contains two main groups of examples, the first set are taken from playing using the strategy from the beginning of the game. However just using these examples can cause ILASP to learn a hypothesis that learns features of the states not the moves. With all of the examples, ILASP was able to learning the following hypothesis[1]:

$$
\begin{aligned}
\mathit{fill\_1}(\mathit{Box2}) &\leftarrow \mathit{adj}(\mathit{Box1}, \mathit{Box2}), \mathit{does}(\_, \mathit{mark}(\mathit{Box1})). \\
\mathit{fill\_2}(\mathit{Box2}) &\leftarrow \mathit{adj}(\mathit{Box1}, \mathit{Box2}), \mathit{box}(\mathit{Box1}, \mathit{Player}), \\
&\qquad \mathit{does}(\mathit{Player}, \mathit{mark}(\mathit{Box2})). \\
&\leftsquigarrow \mathit{fill\_1}(\mathit{Box}), \mathit{leftmost}(\mathit{Box}, \mathit{Score}).[-\mathit{Score}@1, \mathit{Box}, \mathit{Score}] \\
&\leftsquigarrow \mathit{fill\_2}(\mathit{Box}), \mathit{leftmost}(\mathit{Box}, \mathit{Score}).[-\mathit{Score}@2, \mathit{Box}, \mathit{Score}]
\end{aligned}
$$

ILASP has created two predicates, *fill_1* which describes a box that is adjacent to a box that is being marked, and *fill_2* which describes a player marking a box next to one of their own boxes. Putting this together with the weak constraint we get: "at the highest priority gain the reward of how far left a box is if there is already an adjacent box that you have marked, otherwise choose the leftmost box with the most neighbours". Because of the complexity of the game this is equivalent to the correct strategy presented in (Zhang and Thielscher, 2015).

8.2.3 FORWARD-THINKING STRATEGIES

Forward-thinking strategies involve the modal operator ◯ which means "will be true in the next state". Further, this operator can be nested giving rise to strategies that reason several steps into the future, such as the *defence* strategy rule below. In their paper Zhang et al. state they are unable to represent strategies involving ◯ in ASP. In this section, I experiment with using Deep Context Dependent LOAS tasks in order to learn strategies using $\bigcirc$.

Experiment 8.6 (Defence)**.** If the cross player were to play in the first box the dot player would lose using the *thoughtful* strategy, assuming the cross player is rational. This is illustrated in Figure 8.1 below, where the right branch represents the *thoughtful* strategy. It would make more sense for the dot player to block the cross player in order to try and force a draw.

The *defence* strategy means that if there is a move that the opponent could play to win, make that move instead. The learning task described below uses the *deep orderings* from Chapter 7 (Definition 7.2.3). Given the learning task $T_{def} = \langle B, S_M, E^+_O, O\rangle$ where

$$\begin{aligned} B &\quad \text{is the same as in } T_{thoughtful}, \\ M &= \langle \emptyset, \emptyset, M^o, \{-1,1\}, 1\rangle \\ M^o &= \{goal(\mathrm{v}_{role}, \mathrm{c}_{reward}), role(\mathrm{v}_{role}), control(\mathrm{v}_{role})\} \\ O^d &= \{\langle(\cdot, \boxtimes\,\square\,\square\,\square, a_2), (\cdot, \boxtimes\,\square\,\square\,\square, a_3), 2\rangle \end{aligned}$$

The relevant examples are then expanded into a game tree (see Figure 8.1). Along with generated examples, and the injected meta program (described in Section 7.4) ILASP was able to learning the following hypotheses[1]: and at depth 2 you get

$$\leftsquigarrow \text{not } goal(V0,0), control(V0).[-1@1, 1, V0]$$

When applied to minimax at depth two this corresponds to the strategy "make sure you have not lost on your next turn".

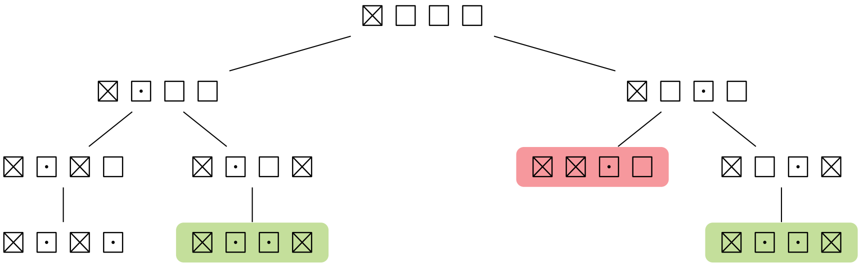


Figure 8.1. Cross-Dot *defence* game tree generated from $\langle(\cdot, \boxtimes\,\square\,\square\,\square, a_2), (\cdot, \boxtimes\,\square\,\square\,\square, a_3), 2\rangle$, the ordering example, with a depth of 2. Green and red represent a winning and losing state for $\cdot$, respectively.

8.3 COMPARISON

In this chapter we have seen that, using weak constraints, ILASP is able to learn the strategies presented by Zhang and Thielscher. The weak constraints have the advantage of using integer weights in order to penalise or reward moves, whilst the logical language described in Zhang and Thielscher (2015) suffer from verbosity, for example in the *leftmost* strategy. Using these weights to create a 'heatmap' of locations is an idea that is revisited in Experiment 10.4.

Zhang and Thielscher translate a subset of strategies written in their modal logic into ASP. Specifically, all strategies using the modal operator $\bigcirc$. They use traditional constraints in order to restrict the set of legal moves down to a few that are considered good with respect to the strategy. This means that if a particular strategy does not satisfy some of the properties shown in the paper (e. g. completeness and determinism) then one could find themselves in a situation where no moves can be recommended, despite legal moves existing. This problem can be quite easily rectified by saying that any legal move is part of the strategy, if rest of the strategy failed. In their modal logic this is written as:

$$\begin{aligned} \mathit{fill\_any}^i &= \bigvee_{a^i \in A^i} \Big(\mathit{legal}(a^i) \wedge \mathit{does}(a^i) \Big) \\ \mathit{strat}^i &= r_1 \triangledown r_2 \triangledown \cdots \triangledown \mathit{fill\_any}^i \end{aligned}$$

Which is to be read as “try all strategies r_i in decreasing order of priority and if they all fail then simply select a legal action”. However, when translated into ASP this becomes:

```
strat(A, T) :- r_1(A, T).
strat(A, T) :- r_2(A, T), not r_1(_, T).
%...
strat(A, T) :- fill_any(A, T), not r_n(_, T).
fill_any(A, T) :- legal(A, T).
```

Whereas when using weak constraints this process is completely omitted as if no weak constraints are violated (i. e. no penalties are gained) then every legal action will by default score 0 and any one can be selected.

Part III

EVALUATION

9 LEARNING THE GAME RULES

Learning the rules of a game is the first step in coming up with a strategy that will allow you to defeat your opponents. The rules introduce you to the board, cards, and pieces of the game. They show you how everything can interact and what actions you have available to you on your turn. Most importantly the rules let you know how to win the game, and by putting all of these components together you can begin to formulate a plan that will allow you to have a better chance of winning.

In this chapter, I will outline the methodologies used in order to learn the rules of *Onitama*, *Five Field Kono* and *Cross-Dot*. Each experiment below lists the examples and mode bias provided to ILASP, in Appendix B you can find the exact options passed to ILASP and what features were used.

9.1 PROCESS

For each game a representation was created in GDL and translated into ASP. In the games looked at here the legal actions available can be calculated directly from the current state of the board. Thus the `next(·)` and `does(·,·)` predicates can be removed from the representation, significantly reducing the grounding of the meta representation.

Each game was translated with as few helper predicates as possible, for example `rows(·)` and `columns(·)` were added to *Onitama* in order to extract rows and columns from a pair of cells. All games had types added to the background knowledge in order to overcome free variables and bound cell ranges etc.. The background knowledge for each game can be found in Appendix A.

Whilst you could achieve the same hypothesis from observing many games being played, examples have been provided by an 'oracle'. This way mimics the way a parent may teach their child a game, additionally it means that examples can be chosen to specifically show a new rule or edge case. In order for ILASP to learn an accurate representation of the rules 'nonsense' moves must be explicitly ruled out as illegal moves. For example, you are unable to move from a empty square to another empty square.

9.2 LEARNING

Experiment 9.1 (Onitama Rules). The task takes the background knowledge from Logic Program A.1 without the action choice rule and the `next(·)` predicate. The mode bias has been built such that instead of creating one rule that defines the legal moves it learns sub predicates each with a restricted search space. This reduces the maximum number of variables that need to be used, and the number of literals that a rule contains.

The mode declaration of this task added some additional rules to the background knowledge as, at the time, ILASP did not support arithmetic expressions in the bias constraints.

$$
\begin{aligned}
&delta(D) \leftarrow card(\_,(D,\_)).\\
&delta(D) \leftarrow card(\_,(\_,D)).
\end{aligned}
$$

$$\begin{aligned}\mathit{math}(B-C,B,C,1) &\leftarrow \mathit{cell}(B-C,B),\mathit{delta}(C).\\ \mathit{math}(B+C,B,C,-1) &\leftarrow \mathit{cell}(B+C,B),\mathit{delta}(C).\end{aligned}$$

The mode declaration used is as follows, the bias constraints used can be found in Section B.1.1, along with the positive and negative examples:

$$M_h = \begin{cases} \mathit{legal}(\mathrm{v}_{\mathit{role}},\mathit{move}(\mathrm{v}_{\mathit{pair}},\mathrm{v}_{\mathit{card}}))\\ \mathit{pred\_1}(\mathrm{v}_{\mathit{role}},(\mathrm{v}_{\mathit{cell}},\mathrm{v}_{\mathit{cell}}))\\ \mathit{pred\_2}(\mathrm{v}_{\mathit{card}},\mathrm{v}_{\mathit{pair}},\mathrm{v}_{\mathit{dir}})\\ \mathit{pred\_3}(\mathrm{v}_{\mathit{role}},\mathrm{v}_{\mathit{cell}})\\ \mathit{pred\_4}(\mathrm{v}_{\mathit{role}},\mathrm{v}_{\mathit{cell}})\\ \mathit{pred\_5}((\mathit{cell}(\mathrm{v}_{\mathit{index}},\mathrm{v}_{\mathit{index}}),\mathit{cell}(\mathrm{v}_{\mathit{index}},\mathrm{v}_{\mathit{index}})),(\mathrm{v}_{\mathit{index}},\mathrm{v}_{\mathit{index}}))\\ \mathit{pred\_6}(\mathrm{v}_{\mathit{delta}},\mathrm{v}_{\mathit{pair}},\mathrm{v}_{\mathit{dir}})\\ \mathit{pred\_7}(\mathrm{v}_{\mathit{delta}},\mathrm{v}_{\mathit{pair}},\mathrm{v}_{\mathit{dir}}) \end{cases}$$

$$M_b = \left\{\begin{array}{ll} \mathit{control}(\mathrm{v}_{\mathit{role}}) & \mathit{pred\_1}(\mathrm{v}_{\mathit{role}},\mathrm{v}_{\mathit{pair}})\\ \mathit{location}(\mathit{pawn}(\mathrm{v}_{\mathit{rank}},\mathrm{v}_{\mathit{role}}),\mathrm{v}_{\mathit{cell}}) & \mathit{pred\_2}(\mathrm{v}_{\mathit{card}},\mathrm{v}_{\mathit{pair}},\mathrm{v}_{\mathit{dir}})\\ \mathit{in\_hand}(\mathrm{v}_{\mathit{role}},\mathrm{v}_{\mathit{card}}) & \mathit{pred\_3}(\mathrm{v}_{\mathit{role}},\mathrm{v}_{\mathit{cell}})\\ \mathit{card}(\mathrm{v}_{\mathit{card}},(\mathrm{v}_{\mathit{delta}},\mathrm{v}_{\mathit{delta}})) & \mathit{pred\_4}(\mathrm{v}_{\mathit{role}},\mathrm{v}_{\mathit{cell}})\\ \mathit{dir}(\mathrm{v}_{\mathit{role}},\mathrm{v}_{\mathit{dir}}) & \mathit{pred\_4}(\mathrm{v}_{\mathit{role}},\mathrm{v}_{\mathit{cell}})\\ \mathit{cell}(\mathrm{v}_{\mathit{index}},\mathrm{v}_{\mathit{index}}) & \mathit{pred\_5}(\mathrm{v}_{\mathit{pair}},(\mathrm{v}_{\mathit{index}},\mathrm{v}_{\mathit{index}}))\\ \mathit{math}(\mathrm{v}_{\mathit{index}},\mathrm{v}_{\mathit{index}},\mathrm{v}_{\mathit{delta}},\mathrm{v}_{\mathit{dir}}) & \mathit{pred\_6}(\mathrm{v}_{\mathit{delta}},\mathrm{v}_{\mathit{pair}},\mathrm{v}_{\mathit{dir}})\\ & \mathit{pred\_7}(\mathrm{v}_{\mathit{delta}},\mathrm{v}_{\mathit{pair}},\mathrm{v}_{\mathit{dir}}) \end{array}\right.$$

Remark. The hypothesis that is learnt can be reduced to one rule which has a smaller grounding by replacing the occurrences of `pred_X(·)` by their body's.

$$\begin{aligned}
\mathit{pred\_7}(DR,(From,To),Dir) &\leftarrow \mathit{math}(Row2,Row1,DR,Dir), \tag{9.1}\\
&\quad\ \mathit{rows}(From,To,(Row1,Row2)).\\
\mathit{pred\_6}(DC,(From,To),Dir) &\leftarrow \mathit{math}(Col1,Col2,DC,Dir), \tag{9.2}\\
&\quad\ \mathit{columns}(From,To,(Col1,Col2)).\\
\mathit{pred\_2}(Card,Coords,Dir) &\leftarrow \mathit{card}(Card,(DC,DR)), \tag{9.3}\\
&\quad\ \mathit{pred\_6}(DC,Coords,Dir),\\
&\quad\ \mathit{pred\_7}(DR,Coords,Dir).\\
\mathit{location}(Role,Cell) &\leftarrow \mathit{location}(\mathit{pawn}(\_,Role),Cell). \tag{9.4}\\
\mathit{pred\_1}(Role,(From,To)) &\leftarrow \mathit{location}(\mathit{pawn}(\_,Role),From), \tag{9.5}\\
&\quad\ \text{not } \mathit{location}(Role,To),\\
&\quad\ \mathit{cell}(To),\mathit{control}(Role).\\
\mathit{legal}(Role,\mathit{move}(Coords,Card)) &\leftarrow \mathit{in\_hand}(Role,Card),\mathit{dir}(Role,Dir), \tag{9.6}\\
&\quad\ \mathit{pred\_1}(Role,Coords),\\
&\quad\ \mathit{pred\_2}(Card,Coords,Dir).
\end{aligned}$$

The hypothesis learnt can be described in English in the following way:

(9.1) *pred_7* is true when there is a correct translation to the rows of the coordinates;
(9.2) *pred_6* is true when there is a correct translation to the columns of the coordinates;
(9.3) *pred_2* is true when there is a card that represents the translation of the pawn;
(9.4) *location* is the projection of the *location* predicate, removing the pawn's rank;
(9.5) *pred_1* is true when the starting location contains *your* pawn and the ending location does not;
(9.6) *legal* is true when a card in your hand provides the translation of the pawn (given your direction) and that you have moved *your* pawn to a valid location.

This hypothesis can be compressed into the following form:

$$\begin{aligned}
\mathit{legal}(\mathit{Role}, \mathit{move}((\mathit{From}, \mathit{To}), \mathit{Card})) \leftarrow\ & \mathit{in\_hand}(\mathit{Role}, \mathit{Card}), \mathit{dir}(\mathit{Role}, \mathit{Dir}),\\
& \mathit{location}(\mathit{pawn}(\_, \mathit{Role}), \mathit{From}),\\
& \text{not } \mathit{location}(\mathit{pawn}(\_, \mathit{Role}), \mathit{To}),\\
& \mathit{cell}(\mathit{To}), \mathit{control}(\mathit{Role}),\\
& \mathit{card}(\mathit{Card}, (\mathit{DC}, \mathit{DR})),\\
& \mathit{math}(\mathit{Col1}, \mathit{Col2}, \mathit{DC}, \mathit{Dir}),\\
& \mathit{columns}(\mathit{From}, \mathit{To}, (\mathit{Col1}, \mathit{Col2})),\\
& \mathit{math}(\mathit{Row2}, \mathit{Row1}, \mathit{DR}, \mathit{Dir}),\\
& \mathit{rows}(\mathit{From}, \mathit{To}, (\mathit{Row1}, \mathit{Row2})).
\end{aligned}$$

Experiment 9.2 (Five Field Kono Rules).

$$\begin{aligned}
&\mathit{adj}(\mathit{cell}(\mathit{Row1}, \mathit{Col1}), \mathit{cell}(\mathit{Row2}, \mathit{Col2})) \leftarrow\\
&\qquad \mathit{cell}(\mathit{Row1}, \mathit{Col1}), \mathit{cell}(\mathit{Row2}, \mathit{Col2}),\\
&\qquad \mathit{Row2} == \mathit{Row1} + 1, \mathit{Col2} == \mathit{Col1} + 1.
\end{aligned} \tag{9.7}$$

$$\begin{aligned}
&\mathit{adj}(\mathit{cell}(\mathit{Row1}, \mathit{Col1}), \mathit{cell}(\mathit{Row2}, \mathit{Col2})) \leftarrow\\
&\qquad \mathit{cell}(\mathit{Row1}, \mathit{Col1}), \mathit{cell}(\mathit{Row2}, \mathit{Col2}),\\
&\qquad \mathit{Col1} == \mathit{Col2} + 1, \mathit{Row1} == \mathit{Row2} - 1.
\end{aligned} \tag{9.8}$$

$$\begin{aligned}
&\mathit{adj}(\mathit{cell}(\mathit{Row1}, \mathit{Col1}), \mathit{cell}(\mathit{Row2}, \mathit{Col2})) \leftarrow\\
&\qquad \mathit{cell}(\mathit{Row2}, \mathit{Col2}), \mathit{cell}(\mathit{Row1}, \mathit{Col1}),\\
&\qquad \mathit{Col2} == \mathit{Col1} + 1, \mathit{Row2} == \mathit{Row1} - 1.
\end{aligned} \tag{9.9}$$

$$\begin{aligned}
&\mathit{adj}(\mathit{cell}(\mathit{Row1}, \mathit{Col1}), \mathit{cell}(\mathit{Row2}, \mathit{Col2})) \leftarrow\\
&\qquad \mathit{cell}(\mathit{Row1}, \mathit{Col1}), \mathit{cell}(\mathit{Row2}, \mathit{Col2}),\\
&\qquad \mathit{Col1} == \mathit{Col2} + 1, \mathit{Row2} == \mathit{Row1} - 1.
\end{aligned} \tag{9.10}$$

$$\begin{aligned}
&\mathit{legal}(\mathit{Role}, \mathit{move}(\mathit{cell}(\mathit{Row1}, \mathit{Col1}), \mathit{cell}(\mathit{Row2}, \mathit{Col2}))) \leftarrow\\
&\qquad \mathit{adj}(\mathit{cell}(\mathit{Row1}, \mathit{Col1}), \mathit{cell}(\mathit{Row2}, \mathit{Col2})), \mathit{control}(\mathit{Role})\\
&\qquad \mathit{state}(\mathit{Row1}, \mathit{Col1}, \mathit{Role}), \mathit{state}(\mathit{Row2}, \mathit{Col2}, e).
\end{aligned} \tag{9.11}$$

The hypothesis learnt can be described in English in the following way:

(9.7)–(9.10) *adj* represents that immediate diagonals are adjacent to the current location (this is more obvious after some arithmetic manipulation);

(9.11) A move is *legal* when the destination is empty and adjacent to the starting location, which has one of *your* pieces.

This hypothesis can be compressed into the following form:

$$\begin{aligned}
&adj(cell(Row, Col), cell(Row+1, Col+1)) \leftarrow cell(Row, Col), cell(Row+1, Col+1).\\
&adj(cell(Row, Col), cell(Row+1, Col-1)) \leftarrow cell(Row, Col), cell(Row+1, Col-1).\\
&adj(cell(Row, Col), cell(Row-1, Col+1)) \leftarrow cell(Row, Col), cell(Row-1, Col+1).\\
&adj(cell(Row, Col), cell(Row-1, Col-1)) \leftarrow cell(Row, Col), cell(Row-1, Col-1).\\
&legal(Role, move(cell(Row1, Col1), cell(Row2, Col2))) \leftarrow\\
&\qquad adj(cell(Row1, Col1), cell(Row2, Col2)), control(Role)\\
&\qquad state(Row1, Col1, Role), state(Row2, Col2, e).
\end{aligned}$$

Experiment 9.3 (Cross-Dot Rules). As the search space of the Cross-Dot game is much smaller I also added the head for learning the *noop* action, i. e. does not move.

$$legal(Role, mark(Box)) \leftarrow box(Box, b), control(Role). \tag{9.12}$$

$$legal(Role, noop) \leftarrow \text{not } control(Role), role(Role). \tag{9.13}$$

10 LEARNING AND EXPRESSING STRATEGIES

Capturing, advancing pawns and forking[1] are all tactics that a player may enforce within a game in order to try and gain the advantage over their opponent. These tactics are also calculable by only looking at the current board position, and they form the building blocks of winning strategies. A lot of powerful strategies in games come from composing smaller strategies that solve particular sub-goals within the game. For example, during the opening you may be looking to capture student pawns in order to restrict the opponents movement, and in the end game look at advancing your master pawn across the board to your opponent's temple square. Later these simple strategies are used in order to create something more complex. As well as being combined they can be reasoned about at different depths of the tree and, in a similar fashion to Chapter 8, we will explore applying deep orderings to player's strategies to learn stronger strategies still.

10.1 IMMEDIATE STRATEGIES

10.1.1 WINNING

Whilst it seems obvious, achieving the winning conditions must be encoded within the strategy. The idea of winning, losing or drawing is also the basis of algorithms such as Monte Carlo tree search, which when they arrive at a winning node propagate a positive value up the game tree, a negative value if it is a losing node or zero for a drawing position. This simple heuristic can be a good indicator of a board position if many games are simulated by randomly walking the tree.

In many games, including *Onitama*, there are multiple ways to win. Recall from the rules (Section 2.1) that in *Onitama* you can either capture your opponent's master, or navigate your own master to your opponent's temple. It may be of interest to learn which of these strategies is preferred and if one method is dominant over another. However, learning the dominant/preferred strategy will require a more complex task, as the best option can often depend on both the cards, and the strategy of your opponent.

Experiment 10.1 (Try to Win)**.** For this experiment we used ILASP2i to perform the learning task as there is no noise. The task given to ILASP was $T = \langle B, S_M, E^+, \emptyset, O^b, \emptyset \rangle$ where,

B is the logic program from Logic Program A.1

$M = \langle \emptyset, \emptyset, \{goal(\mathrm{v}_{role}, \mathrm{c}_{score}), control(\mathrm{v}_{role})\}, \{-1, 1\}, 1 \rangle$

E^+ is the set of examples, e_i, with the contexts, C_i, below and empty inclusions and exclusions

$O_b = \{\langle e_1, e_2 \rangle, \langle e_1, e_3 \rangle\}$

Note. For this task we can take advantage of the built in GDL specified predicate *goal*.

---

[1] Threatening to capture two pawns with one

```
% Context: C1
location(pawn(student,red),cell(5,5)).
location(pawn(student,blue),cell(4,3)).
location(pawn(student,red),cell(4,5)).
location(pawn(master,red),cell(2,1)).
location(pawn(student,red),cell(2,2)).
location(pawn(student,red),cell(2,4)).
location(pawn(student,blue),cell(1,1)).
location(pawn(student,blue),cell(1,5)).
control(blue).
in_play(monkey).
in_play(ox).
in_play(sable).
in_hand(red,monkey).
in_hand(blue,ox).

% Context: C2
location(pawn(student,blue),cell(4,3)).
location(pawn(student,red),cell(4,4)).
location(pawn(student,red),cell(4,5)).
location(pawn(student,red),cell(3,5)).
location(pawn(master,red),cell(2,1)).
location(pawn(student,red),cell(2,2)).
location(pawn(master,blue),cell(2,4)).
location(pawn(student,blue),cell(1,1)).
location(pawn(student,blue),cell(1,5)).
control(blue).
in_play(monkey).
in_play(ox).
in_play(sable).
in_hand(red,monkey).
in_hand(blue,ox).

% Context: C3
location(pawn(student,red),cell(5,5)).
location(pawn(student,blue),cell(4,3)).
location(pawn(student,red),cell(3,4)).
location(pawn(student,red),cell(3,5)).
location(pawn(master,red),cell(2,1)).
location(pawn(student,red),cell(2,2)).
location(pawn(master,blue),cell(2,4)).
location(pawn(student,blue),cell(1,1)).
location(pawn(student,blue),cell(1,5)).
control(blue).
in_play(monkey).
in_play(ox).
in_play(sable).
in_hand(red,monkey).
in_hand(blue,ox).
```

The hypothesis learnt by ILASP is:

$$\leadsto goal(Role, 100), \text{not } control(Role).[-1@1, Role]$$

At first this result may look odd as it says "a reward is given when the player *not* in control reaches the goal of 100", however when looking at the states it is because the player who has just moved is no longer having their turn.

10.1.2 CAPTURING PIECE

A common strategy among *Onitama* beginners is to capture a piece when they can, and whilst this strategy can sometimes be short-sighted (as seen in Chapter 7) it is important to be able to learn the strategy and represent it in the background knowledge.

Intuitively to humans a hypothesis of 'capturing' could be thought of as "moving into a square with containing opponent pawn", This contains the notions of 'moving', 'location' and 'opponent pawns'. Therefore predicates relating to these concepts would be useful to add to the mode declarations. With this in mind we present the following experiment.

Experiment 10.2 (Capture)**.** Using the digital versions of the games many examples of capturing a pawn if at all possible (and randomly choosing in the case of multiple possibilities) were collected. For this experiment we used ILASP 2i in order to do the learning as there is no noise. The task given to ILASP was $T = \langle B, S_M, E^+, \emptyset, O^b, \emptyset \rangle$ where,

B is the logic program from Logic Program A.1

$$M = \left\langle \emptyset, \emptyset, \left\{ \begin{array}{l} location(pawn(\mathrm{v}_{rank}, \mathrm{v}_{role}), \mathrm{v}_{cell}), \\ control(\mathrm{v}_{role}), does(\mathrm{v}_{role}, move(\mathrm{v}_{cell}, \mathrm{v}_{cell})) \end{array} \right\}, \{-1, 1\}, 1 \right\rangle$$

E^+ is the set of examples, e_i, with the contexts, C_i, below and empty inclusions and exclusions $O_b = \{\langle e_1, e_2\rangle, \langle e_3, e_4\rangle\}$

```
% Context: C1
location(pawn(student,red),cell(5,1)).
location(pawn(student,red),cell(5,3)).
location(pawn(master,red),cell(4,2)).
location(pawn(student,blue),cell(3,2)).
location(pawn(student,blue),cell(3,5)).
location(pawn(student,blue),cell(2,2)).
location(pawn(master,blue),cell(1,3)).
location(pawn(student,blue),cell(1,5)).
control(red).
in_play(sable).
in_play(ox).
in_play(monkey).
in_hand(red,sable).
in_hand(blue,ox).
```

```
% Context: C2
location(pawn(student,red),cell(5,1)).
location(pawn(student,red),cell(5,3)).
location(pawn(master,red),cell(4,2)).
location(pawn(student,blue),cell(3,2)).
location(pawn(student,blue),cell(3,3)).
location(pawn(student,red),cell(3,5)).
location(pawn(student,blue),cell(2,2)).
location(pawn(master,blue),cell(1,3)).
location(pawn(student,blue),cell(1,5)).
control(red).
in_play(sable).
in_play(ox).
in_play(monkey).
in_hand(red,sable).
in_hand(blue,ox).
```

```
% Context: C3
location(pawn(student,red),cell(5,1)).
location(pawn(student,red),cell(5,5)).
location(pawn(student,blue),cell(4,2)).
location(pawn(student,blue),cell(3,5)).
location(pawn(student,blue),cell(2,2)).
location(pawn(master,blue),cell(1,3)).
location(pawn(student,blue),cell(1,5)).
control(red).
in_play(monkey).
in_play(sable).
in_play(ox).
in_hand(red,monkey).
in_hand(blue,sable).
```

```
% Context: C4
location(pawn(student,red),cell(5,1)).
location(pawn(student,red),cell(5,5)).
location(pawn(master,red),cell(4,2)).
location(pawn(student,blue),cell(3,2)).
location(pawn(student,blue),cell(3,5)).
location(pawn(student,blue),cell(2,2)).
location(pawn(master,blue),cell(1,4)).
location(pawn(student,blue),cell(1,5)).
control(red).
in_play(monkey).
in_play(sable).
in_play(ox).
in_hand(red,monkey).
in_hand(blue,sable).
```

The hypothesis learnt by ILASP is:

$$\leftsquigarrow \mathit{location}(\mathit{pawn}(\mathit{Rank}, \mathit{Player}), \mathit{Cell}).[1@1, \mathit{Rank}, \mathit{Player}, \mathit{Cell}]$$

This hypothesis does not contain the notion of movement and has a penalty instead of a reward! On closer inspection it can be seen that this weak constraint is attempting to minimise the number of unique pawns on the board, as you are unable to capture your own pieces this translates to capturing the opponent.

This experiment is not particularly interesting, it has a single predicate in the body and a single priority. Also consider the possibility of having the choice to capture a master piece or a student piece, the hypothesis above would weight them the same. In order to express this preference we can change the mode declaration by replacing v_{rank} with c_{rank}, and adding the following ordering example: $O^{b'} = O^b \cup \{\langle e_5, e_6\rangle\}$

```
% Context: C5
location(pawn(student,red),cell(3,4)).
location(pawn(student,red),cell(5,5)).
location(pawn(student,blue),cell(2,5)).
location(pawn(student,blue),cell(3,3)).
location(pawn(student,blue),cell(2,1)).
location(pawn(master,blue),cell(1,3)).
control(red).
in_play(boar).
in_play(sheep).
```

```
in_play(eel).
in_hand(red,boar).
in_hand(blue,eel).

% Context: C_6
location(pawn(student,red),cell(5,5)).
location(pawn(master,red),cell(3,3)).
location(pawn(student,blue),cell(3,4)).
location(pawn(student,blue),cell(3,2)).
location(pawn(student,blue),cell(2,1)).
location(pawn(master,blue),cell(1,3)).
control(red).
in_play(boar).
in_play(sheep).
in_play(eel).
in_hand(red,boar).
in_hand(blue,eel).
```

Experiment 10.3 (Capture: Master or Student?)**.** Given the new task $T = \langle B, S_{M'}, E^{+}, \emptyset, O^{b'}, \emptyset \rangle$, based on the task in Experiment 10.2 above, where

$$M' = \langle \emptyset, \emptyset, \{location(pawn(\mathrm{c}_{rank}, \mathrm{v}_{role}), \mathrm{v}_{cell}), control(\mathrm{v}_{role})\}, \{-1, 1, -2, 2\}, 1 \rangle$$

notice how extra weights have been added, alternatively we could have chosen to allow two priority levels. They have been added as one might expect at a high priority the strategy describes the master and at a lower priority, the students.

ILASP learns the hypothesis:

$$\begin{aligned} \leadsto\ & location(pawn(master, Player), Cell1), \\ & location(pawn(student, Player), Cell2).[1@1, Player, Cell1, Cell2] \end{aligned}$$

The immediate observation is that ILASP has been able to cover the ordering examples using a single rule. This hypothesis means the following: *for each player penalise each of their students when their master is also on the board.* In a scenario when you cannot capture the master, then the masters will still be on the board next turn and so this strategy is equivalent to before. However, when you can capture the opponent's master it will no longer be on the board and so only the winning player's students are counted.

10.1.3 SPACE ADVANTAGE

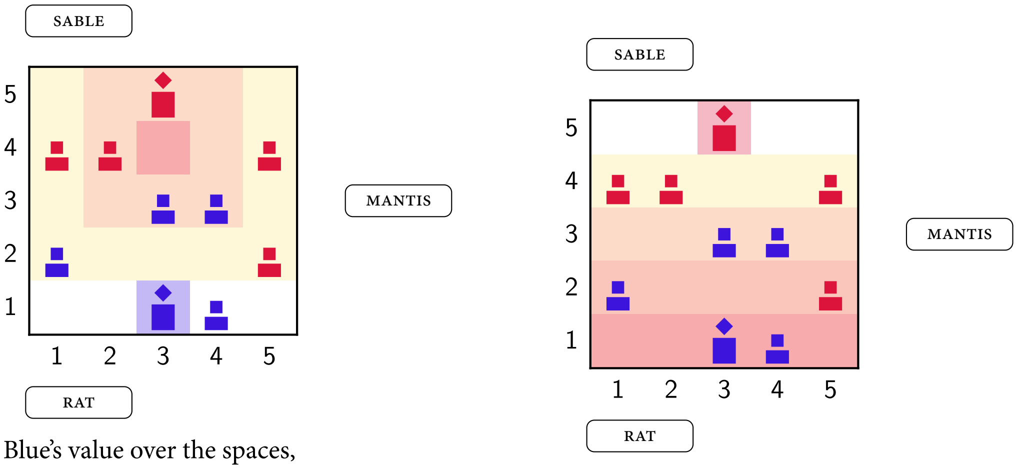
Experiment 10.4 (Controlling Regions of the Board)**.** A common tactic in (certain) abstract strategy games is to control as much space a possible (or certain spaces of the board). In chess, controlling the centre is often touted as good practice. It can free up paths for your bishops, and allow knights to control more of the important squares. Figure 10.2 shows two illustrations, on the left is a board state with blue's control shaded, and on the right is the same for red. We can see that blue is making it very difficult for red to advance. Red is only controlling their half of the board, meanwhile blue is moving pawns up across the board, creating lots of space to move into.

This experiment used a large set of noisy examples collected from full games but was unable to learn a suitable strategy, due to the noise in the examples and mode declarations not providing enough information.

Experiment 10.5. In this experiment the examples are generating from a player correcting the decisions of an AI. These examples were collected using the *training-mode* of digital *Onitama*, see Section 6.1.1 for details. The task was run using ILASP2i because we assume there is no noise as the player is explicitly correcting a move.

The mode declaration for the task is:

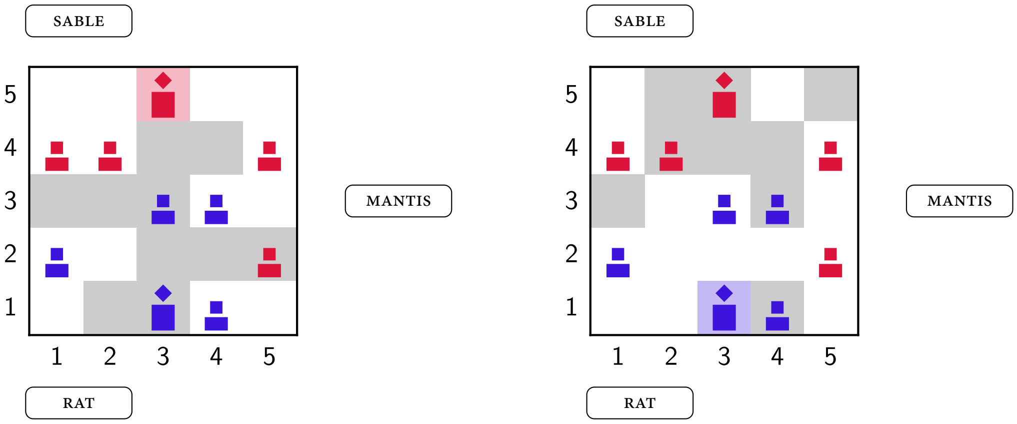
$$M = \left\langle \emptyset, \emptyset, \left\{ \begin{array}{l} valid\_translation(\_, \mathrm{v}_{cell}, \_, \mathrm{v}_{role}), \\ control(\mathrm{v}_{role}), opponent(\mathrm{v}_{role}, \mathrm{v}_{role}), \\ location(pawn(\mathrm{c}_{rank}, \mathrm{v}_{role}), \mathrm{v}_{cell}) \end{array} \right\}, \{-1, 1\}, 1 \right\rangle$$



(a) Blue's value over the spaces, i. e. approaching the spaces near the temple is a good move

(b) Red's valuation over the spaces, i. e. moving forward is a good move

Figure 10.1. Illustrations of how valuing different areas of the board affects strategy. Red represents higher valuation and white is the lowest.



(a) Blue's control over the board

(b) Red's control over the board

Figure 10.2. Illustrations of how controlling space on a board can give be advantageous. Grey areas represent spaces that the cards allow each player to move to. The strategies learnt in Experiment 10.5 and Experiment 10.7 both have this concept in their mode declarations ($valid\_translation(\cdot, \cdot, \cdot, \cdot)$)

The contexts used to create the example are as follows

The learnt hypothesis learnt is:

$$\leftsquigarrow \text{not } valid\_translation(\_, To, \_, Player), location(pawn(master, Player), To).[1@1, To, Player]$$

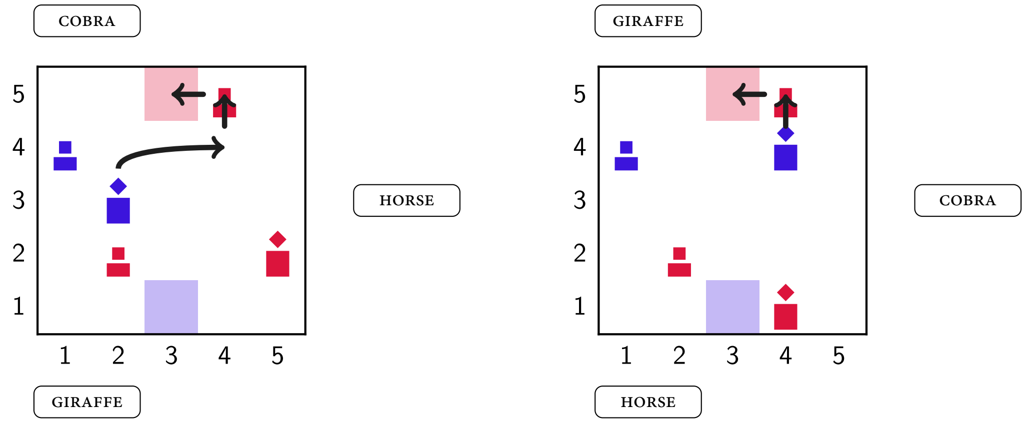
$$\leftsquigarrow \text{not } valid\_translation(\_, To, \_, Player), location(pawn(master, Enemy), To),$$
$$control(Enemy), opponent(Player, Enemy).[1@1, To, Player, Enemy]$$

This hypothesis says that you get penalised for not being able to defend your master pawn. The second rule says that if one is not able to threaten the opponent's master then a penalty is received. Note that the `control(·)` here is after the move has been made and so this refers to the opponent's master.

10.2 COMPLEX STRATEGIES

Experiment 10.6 (Planning a Path for the Master). Many chess engines (e. g. *Stockfish*) use endgame tables (e. g. Nalimov endgame tablebases[2]) towards the end of the game, normally when there are fewer than six pieces on the board, as these small portions of the sub-trees have been solved. A wrong move can quickly turn a winning position into a drawing position (or worse a losing position). In *Onitama* because of the fixed moves you can find yourself in a situation where providing you notice it early enough you can force a win by navigating the master correctly to the temple square, this is occurs more often in the three card variant. Learning this from examples is something that could be done using enough forethought, and is a good test of how powerful the deep orderings are, and what they can achieve. Figure 10.3 demonstrates a series of moves for the blue master ♔ which red is unable to defend against. This experiment is an extension of the "try to win" strategy (Experiment 10.1) involving deep orderings.

While in theory reasoning 5 moves into the future is possible the $ILP^{context}_{\textsc{loas}}$ with $\mathcal{Q}$ that is generated contains 1.15×10^7 brave orderings for the leaves. This is due to the branching factor of the game. As the task has to be fully described before being solved by ILASP there currently is no way of partially evaluating the tree. In out future work (Section 11.2) we discuss one method of pruning the tree by evaluating 'quiescent nodes' (see Section 3.5.3). For this reason the task below uses a deep ordering based on Figure 10.3b which only has 24 649 brave orderings, meaning an average branching factor of 5.39 moves.



(a) Reasoning 5 moves into the future **(b)** Reasoning 3 moves into the future

Figure 10.3. Guaranteed win by following the moves shown

We use ILASP 3 with injection in order to encode the following task: $T = \langle B, S_M, E, O, \lhd \rangle$ where,

$$M = \langle \emptyset, \emptyset, \{goal(\mathrm{v}_{role}, \mathrm{v}_{reward}), control(\mathrm{v}_{role})\}, \{\mathrm{v}_{reward}\}, 1\rangle \tag{10.1}$$

$$E^+ = \{e_1, e_2\} \tag{10.2}$$

$$O^d = \{\langle e_1, e_2, 3, \lhd \rangle \tag{10.3}$$

---

[2] `http://www.k4it.de/index.php?topic=egtb&lang=en`

```
% Context: C1
location(pawn(student,red),cell(2,2)).
location(pawn(student,red),cell(5,4)).
location(pawn(master,red),cell(2,5)).
location(pawn(student,blue),cell(4,1)).
location(pawn(master,blue),cell(3,2)).
control(blue).
in_play(giraffe).
in_play(cobra).
in_play(horse).
in_hand(red,cobra).
in_hand(blue,giraffe).
:- not does(blue, move(cell(3,2), cell(4,4),
 ↪   giraffe)).
```

```
% Context: C2
location(pawn(student,red),cell(2,2)).
location(pawn(student,red),cell(5,4)).
location(pawn(master,red),cell(2,5)).
location(pawn(student,blue),cell(4,1)).
location(pawn(master,blue),cell(3,2)).
control(blue).
in_play(giraffe).
in_play(cobra).
in_play(horse).
in_hand(red,cobra).
in_hand(blue,giraffe).
:- not does(blue, move(cell(4,1), cell(3,1),
 ↪   giraffe)).
```

This lead to ILASP generating the hypothesis:

$$\leftsquigarrow goal(Player, Reward).[-Reward@1, Player, Reward]$$

Despite this being a similar hypothesis to the one originally learnt in Experiment 10.1, given the two examples e_1 and e_2 a normal context-dependent LOAS task would have returned this as *unsatisfiable* because of the forward thinking. Next we describe a more complex use case for deep orderings.

Experiment 10.7 (Defending Pawns). In *Onitama* there is very little space to manoeuvre, however it can still be the case that a pawn is left stranded on the board. Take Figure 10.2 as an example, the red student ♙ on $(2,5)$ is left without defence (note the lack of shading from red's perspective).

Unlike chess where the pieces have fixed movement, in a game of *Onitama* the cards will rotate. Therefore to think about defending your pieces you need to calculate both where your pieces will be and what moves they will have. In order to achieve this as part of an ILASP learning task, you can encode the preference as a deep ordering looking at depth 2. As if the other player takes your pawn then you have a chance to capture back (giving a better valuation than just losing a pawn).

This task was created using *training-mode*, with one alteration: a brave to a deep ordering, of depth 1 i.e. considering the opponent's next turn. The mode declaration of the tasks is the same as in Experiment 10.5, the main difference is in this experiment we are looking to defend our own pawns instead of avoid capture completely. For ease, M is restated below:

$$M = \left\langle \emptyset, \emptyset, \left\{ \begin{array}{l} valid\_translation(\_, \mathrm{v}_{cell}, \_, \mathrm{v}_{role}), \\ control(\mathrm{v}_{role}), opponent(\mathrm{v}_{role}, \mathrm{v}_{role}), \\ location(pawn(\mathrm{c}_{rank}, \mathrm{v}_{role}), \mathrm{v}_{cell}) \end{array} \right\}, \{-1, 1\}, 1 \right\rangle$$

The contexts for the partial interpretations using the deep ordering are below, all other contexts used in this task can be found in Section B.1.2.

```
% Context: C7
location(pawn(master,red),cell(5,3)).
location(pawn(student,red),cell(5,4)).
location(pawn(student,red),cell(4,1)).
location(pawn(student,red),cell(4,4)).
location(pawn(student,red),cell(4,2)).
location(pawn(student,blue),cell(2,1)).
location(pawn(student,blue),cell(2,4)).
location(pawn(student,blue),cell(1,1)).
location(pawn(master,blue),cell(1,3)).
```

```
location(pawn(student,blue),cell(1,4)).
control(red).
in_play(rat).
in_play(crab).
in_play(mantis).
in_hand(red,mantis).
in_hand(blue,crab).
:- not does(red, move(cell(4,2), cell(3,1),
 ↪   mantis)).

% Context: C8
location(pawn(master,red),cell(5,3)).
location(pawn(student,red),cell(4,1)).
location(pawn(student,red),cell(4,2)).
location(pawn(student,red),cell(4,4)).
location(pawn(student,red),cell(5,4)).
location(pawn(student,blue),cell(2,1)).
location(pawn(student,blue),cell(2,4)).
location(pawn(student,blue),cell(1,1)).
location(pawn(master,blue),cell(1,3)).
location(pawn(student,blue),cell(1,4)).
control(red).
in_play(rat).
in_play(crab).
in_play(mantis).
in_hand(red,mantis).
in_hand(blue,crab).
:- not does(red, move(cell(5,4), cell(4,5),
 ↪   mantis)).
```

The brave ordering examples, O^b, are:

$$\langle e_1, e_2 \rangle \qquad \langle e_3, e_4 \rangle \qquad \langle e_5, e_6 \rangle$$

and the deep ordering example: $\langle e_7, e_8, 1, \lhd$, a translation of this task can be found in Logic Program B.2.

ILASP learnt the following hypothesis:

$$\leftsquigarrow \text{not } valid\_translation(\_, To, \_, Player), location(pawn(student, Player), To).[1@1, Player, To]$$

This hypothesis translates into English as: "minimise the number of students one has that you cannot move to" which means that if your piece were to be captured, you could move to that square and capture back.

10.3 TOURNAMENTS

In order to test whether or not the hypotheses we have learnt in this chapter are usable in a game and able to perform well we pitted strategies against each other and against a random (no preference) and hand coded strategy. Each experiment involves 100 simulated games of two strategies, the random nature of the game means that each AI gets to start roughly evenly. We then perform a binomial, two-tailed hypothesis test to see whether or not the strategies are equal or if there is a bias towards one. Table 10.1 shows the p-values of the various tests, as you can see most of the experiments show that we should reject the hypothesis that the strategies are equal.

10.4 SUMMARY

In this section we give a high level overview of the learning tasks in the project and comment on what our results show. Table 10.2 shows the hypotheses learnt across the various chapters of this report.

STRATEGY 1 VS. STRATEGY 2	P-VALUE	RESULT
Win or Capture vs. Random	$1.9114 \times 10^{-15}\star$	Win or Capture is clearly a better strategy than random.
Win or Capture vs. Avoid Capture	0.67181	These strategy's are roughly equivalent, we should not reject the null hypothesis
Win or Capture vs. Defend Pieces	$0.01203\star$	The Defend Pieces strategy is considered the better strategy

Table 10.1. Hypothesis tests between various strategies,$\star$ represents a significant value

EXPERIMENT	HYPOTHESIS	NOTES
Experiment 10.1	$\leftsquigarrow goal(Player, 100).[-1@1, Player]$	Useful later when combined with other strategies
Experiment 10.2	$\leftsquigarrow location(pawn(Rank, Player), Cell).$ $[1@1, Rank, Player, Cell]$	Instance of the penalty being used to minimise a board feature
Experiment 10.3	$\leftsquigarrow location(pawn(master, Player), Cell1),$ $location(pawn(student, Player), Cell2).[1@1, Player, Cell1, Cell2]$	
Experiment 10.4	—	This experiment was unsuccessful based on numerous example games, this could be due to noise
Experiment 10.5	$\leftsquigarrow \text{not } valid\_translation(\_, To, \_, Player), location(pawn(master, Player), To).[1@1, To, Player]$ $\leftsquigarrow \text{not } valid\_translation(\_, To, \_, Player), location(pawn(master, Enemy), To),$ $control(Enemy), opponent(Player, Enemy).[1@1, To, Player, Enemy]$	
Experiment 10.6	$\leftsquigarrow goal(Player, Reward).[-Reward@1, Player, Reward]$	This was a similar task to Experiment 10.1, but using deep orderings
Experiment 10.7	$\leftsquigarrow \text{not } valid\_translation(\_, To, \_, Player),$ $location(pawn(student, Player), To).[1@1, Player, To]$	
Experiment 8.1	$\leftsquigarrow does(x, mark(Box1)), adj(Box1, Box2), box(Box2, x).[-1@1, Box1, Box2]$	
Experiment 8.2	$\leftsquigarrow does(x, mark(Box1)), adj(Box1, Box2), true(box(Box2, b)).[-1@1, Box1, Box2]$	
Experiment 8.3	$\leftsquigarrow does(x, mark(Box1)), adj(Box1, Box2), true(box(Box2, x)).[-1@2, Box1, Box2]$ $\leftsquigarrow does(x, mark(Box1)), adj(Box1, Box2), true(box(Box2, b)).[-1@1, Box1, Box2]$	
Experiment 8.4	$\leftsquigarrow does(x, mark(Box1)).[Box1@1, Box1]$	
Experiment 8.5	(see page 77)	
Experiment 8.6	$\leftsquigarrow \text{not } goal(V0, 0), control(V0).[-1@1, 1, V0]$	

Table 10.2. Summary of experiments performed throughout the report

In this chapter we have presented a series of experiments created from constructed situations in order to evaluate the feasibility of using weak constraints to learn strategies in games. We started with single predicate weak constraints to assess the winning conditions, afterwards we moved onto looking at basic tactics such as capturing, and combining the two strategies in order to create a more robust strategy. This strategy has shown to be effective in practice, but not as strong as some of the later positional and forward-thinking strategies, just as one might expect.

We then looked into strategies that assign value to positions on the board, utilising weights of weak constraints as a simple way of summing over the pawns on the board.

Finally, we look into applying the deep orderings in order to calculate positions from the given states, just as players do when playing games. We have shown that for smaller depths these tasks work well, and are superior in practice to the immediate strategies.

11 CONCLUSION

Learning from weak constraints means we can express strategies using English once they have been learnt. This allows us to no only understand in more depth what has been learnt but also it allows us to learn more about a game from the strategies of better players. Weak constraints allow us to build upon existing strategies by assigning a priority level to each rule. Using *deep orderings* we are now able to learn about a state based one preferences within the game tree using our ILP^{deep}_{LOAS} tasks.

Attempting to learn from full games, i. e. comparing the move a player made to other possible moves per turn, *did not yield interesting or powerful strategies*. Further experimentation looking into reducing the grounding and constructing a hypothesis space that is better suited to the games could be an option.

Learning only from suggestions provided by a human player allowed us to create a smaller set of examples that we were more certain reflected the strategy of the player. The results that came from these experiments *did show that ILASP could learn strategies that are good in competition*. Having fewer examples means that the tasks can run quickly, this is important for future work in order to incorporate the learning into an *online* AI player.

Learning from *deep orderings* has taken advantage of being able to inject additional meta-programs into ILASP allowing us to create rules which activate only certain learning examples. In this report we have shown proof-of-concept tasks that highlight the added expressiveness of deep orderings. Though, there is further work that needs to be done in order to reduce the size of the tasks for larger depths.

In conclusion, the experimentation in this report has demonstrated that weak constraints are fully capable to learn strategies; what is more, it has shown the explainable nature of the hypotheses learnt.

11.1 ACHIEVEMENTS

1. We have described a method of applying the Game Description Language to learning strategies (Chapter 5). In particular it means that we can use its defined structure in order to construct tree-based examples from the answer sets of a turn. It also means that the framework presented is not fixed to any of the games presented here in this report, the background knowledge for *Onitama*, *Five Field Kono* and *Cross-Dot* all conform to this specification.

2. In Chapter 6, we created a program to simulate games using different types of players (e. g. human, Monte Carlo and the Clingo AIs) in both *tournament-mode* and *training-mode*. In the two modes we can record different types of learning examples that are given directly to ILASP. Again, new types of board games can be added to this program in a simple manner.

3. We present ILP^{deep}_{LOAS}, a new Inductive Logic Programming framework that introduces *deep orderings*. A new way of describing preferences over *possible* futures based on a single move based on the minimax theorem.

4. Following from a review of the literature (Chapter 4) we explored strategies for *Cross-Dot* (see (Zhang and Thielscher, 2015)).
5. To evaluate the learning methods we provide experimentation in Chapter 10 as well as running tournaments (Section 10.3) to compare the strategies that have been learnt.

11.2 FUTURE WORK

Whilst experimenting with ILASP and *weak constraints* we came across many points to explore, such as:

- Extending the hypothesis space to allow custom weights for rules, this would allow *features* of the games to be learnt and added as rules to background knowledge and used in the mode declarations with a length greater than one in order to not bias the mode declaration too much in their favour,
- Extending the ILP^{deep}_{LOAS} to perform a Monte Carlo style search, this would involve only considering a subset of the children that are considered to be 'better' using some heuristic — perhaps learnt or represented using weak constraints.
- Experiment with simple games involving non-determinism, for example the *Royal Game of Ur*, a simple race game using a 4-sided die where the strategy lies in choosing your pieces correctly to hinder your opponent's movement.
- Looking into relating our explanation condition with formal strategy and game logic, such as those in Benthem (2011) and Zhang and Thielscher (2015).

However, there are two more significant pieces of work that could be explored whilst using *weak constraints* to learn strategies. The first is learning features of the game to perform a quiescent search, and the second is effectively identifying examples that encode the strategy.

11.2.1 PERFORMING QUIESCENT SEARCH WITH WEAK CONSTRAINTS

As the minimax tree can explode rapidly, *deep orderings* become infeasible after a certain depth, depending on the average branching factor of the game. In order to cut down on the branches that are expanded an heuristic could be applied to a state and any branches that are deemed to not be of use can be pruned from the tree. This heuristic can be a set of weak constraints that have been learned previously. Clingo has an option to only return answer sets below a certain penalty, this would potentially require using a wider set of weights when learning than done so in this report. In order to not bias future tasks, you could prune the branch with a certain probability, similar to the exploitation/exploration techniques used in Reinforcement Learning.

The weak constraints learnt for this purpose may be more 'feature-based', i. e. a set for captures, a set for checks etc., mimicking the capture-trees and endgame tables in Chess engines. Performing this selective search after a fixed depth minimax is the basis of a quiescent search, and not only prunes the tree but it is able to mitigate against the horizon effect.

11.2.2 IDENTIFYING EXAMPLES WITH STRONG STRATEGIC CHOICES

Based on the results from the experiments we have run in this report we see that having lots of noise and positions where no interesting tactical decisions have been made make learning the strategies more challenging. However, using a human player means that collecting examples is expensive and cumbersome. One future extension to the example collection mechanism could be to have a method of selecting which examples taken from a game are worth keeping and which are not. This could be achieved through a hybrid logic-neural solution using a trained neural network to select examples from a game that are key in different decision processes. The logic framework would offer the same accurate and explainable properties that are so desired in an AI that we have seen throughout this report.

Part IV

APPENDIX

A LOGIC PROGRAMS

Logic Program A.1. Onitama Background Knowledge

```
% Game Constants

cell(5, 1).  cell(5, 2).  cell(5, 3).  cell(5, 4). cell(5, 5).
cell(4, 1).  cell(4, 2).  cell(4, 3).  cell(4, 4). cell(4, 5).
cell(3, 1).  cell(3, 2).  cell(3, 3).  cell(3, 4). cell(3, 5).
cell(2, 1).  cell(2, 2).  cell(2, 3).  cell(2, 4). cell(2, 5).
cell(1, 1).  cell(1, 2).  cell(1, 3).  cell(1, 4). cell(1, 5).

cell(cell(1..5, 1..5)).

temple(red, cell(5,3)).
temple(blue, cell(1,3)).

opponent(red, blue).
opponent(blue, red).

role(red).
role(blue).

dir(red, -1).
dir(blue, 1).

captured_master(Player) :- role(Player), not true(location(pawn(master, Player),
 ↪  _)).

% Action Generation
0 { does(Player, Action) } 1 :- legal(Player, Action), not terminal.

%:- does(Player, A1), does(Player, A2), A1 != A2.

:- does(P, noop), does(P, move(_, _, _)).
% Try splitting this up
:- does(P, move(C1, _, _)), does(P, move(C2, _, _)), C1 < C2.
:- does(P, move(_, C1, _)), does(P, move(_, C2, _)), C1 < C2.

:- legal(Player, _), not terminal, not does(Player, _).


% Game State

% Pawn location
next(location(pawn(Rank, Player), Cell)) :-
  true(location(pawn(Rank, Player), Cell)),
  not does(_, move((_, Cell), _)), not does(Player, move((Cell, _), _)).
```

```
next(location(pawn(Rank, Player), To)) :- true(location(pawn(Rank, Player), From)), does(Player, move((From, To), _)).

% Card in center
true(center_card(Card)) :- in_play(Card), not true(in_hand(red, Card)), not true(in_hand(blue, Card)).

% Player's hand
next(in_hand(Player, Card)) :- true(in_hand(Player, Card)), not true(control(Player)).
next(in_hand(Player, Card)) :- true(center_card(Card)), true(control(Player)).
% Only needed for 5 card variant
next(in_hand(Player, Card)) :- true(in_hand(Player, Card)), Card != Card2, does(Player, move(_, Card2)).

% Player's turn
next(control(Player)) :- true(control(Opp)), opponent(Player, Opp).
next(control(blue)) :- true(control(red)).

goal(Player, 100) :- captured_master(Opp), opponent(Player, Opp).
goal(Player, 100) :- temple(Opp, Temple), true(location(pawn(master, Player), Temple)), opponent(Player, Opp).
goal(Player, 0) :- goal(Opp, 100), opponent(Player, Opp).

terminal :- goal(_, 100).

legal(Player, noop) :- role(Player), not true(control(Player)).
legal(Player, move(cell(FromR, FromC), cell(ToR, ToC), Card)) :-
  true(control(Player)),
  true(location(pawn(_, Player), cell(FromR, FromC))),
  not true(location(pawn(_, Player), cell(ToR, ToC))),
  dir(Player, D), card(Card, (DC, DR)),
  true(in_hand(Player, Card)),
  ToR == FromR-(DR*D), ToC == FromC-(DC*D).
% Cards

card(tiger,(0,-2)).  card(tiger,(0,1)).
card(crab,(0,-1)).  card(crab,(-2,0)).  card(crab,(2,0)).
card(monkey,(1,-1)).  card(monkey,(1,1)).  card(monkey,(-1,1)).  card(monkey,(-1,-1)).
card(crane,(0,-1)).  card(crane,(-1,1)).  card(crane,(1,1)).
card(mantis,(-1,-1)).  card(mantis,(1,-1)).  card(mantis,(0,1)).
card(boar,(0,-1)).  card(boar,(-1,0)).  card(boar,(1,0)).
card(dragon,(-1,1)).  card(dragon,(1,1)).  card(dragon,(2,-1)).  card(dragon,(-2,-1)).
card(elephant,(-1,-1)).  card(elephant,(-1,0)).  card(elephant,(1,0)).  card(elephant,(1,-1)).
card(eel,(-1,-1)).  card(eel,(-1,1)).  card(eel,(1,0)).
card(goose,(-1,-1)).  card(goose,(1,1)).  card(goose,(1,0)).  card(goose,(-1,0)).
card(frog,(-1,-1)).  card(frog,(1,1)).  card(frog,(-2,0)).
card(horse,(0,-1)).  card(horse,(-1,0)).  card(horse,(0,1)).
```

```
card(rabbit,(1,-1)).  card(rabbit,(-1,1)).  card(rabbit,(2,0)).
card(ox,(0,-1)).  card(ox,(1,0)).  card(ox,(0,1)).
card(cobra,(1,-1)).  card(cobra,(1,1)).  card(cobra,(-1,0)).
card(rooster,(1,-1)).  card(rooster,(-1,1)).  card(rooster,(-1,0)).  card(rooster,(1,0)).
card(kirin,(-1,-2)).  card(kirin,(1,-2)).  card(kirin,(0,2)).
card(turtle,(-2,0)).  card(turtle,(-1,1)).  card(turtle,(1,1)).  card(turtle,(2,0)).
card(giraffe,(-2,-1)).  card(giraffe,(2,-1)).  card(giraffe,(0,1)).
card(phoenix,(-2,0)).  card(phoenix,(-1,-1)).  card(phoenix,(1,-1)).  card(phoenix,(2,0)).
card(otter,(-1,-1)).  card(otter,(1,1)).  card(otter,(2,0)).
card(viper,(-2,0)).  card(viper,(0,-1)).  card(viper,(1,1)).
card(iguana,(-2,1)).  card(iguana,(0,-1)).  card(iguana,(1,1)).
card(rat,(-1,0)).  card(rat,(0,-1)).  card(rat,(1,1)).
card(bear,(-1,-1)).  card(bear,(0,-1)).  card(bear,(1,1)).
card(dog,(-1,-1)).  card(dog,(-1,0)).  card(dog,(-1,1)).
card(sable,(1,-1)).  card(sable,(-1,1)).  card(sable,(-2,0)).
card(seasnake,(2,0)).  card(seasnake,(0,-1)). card(seasnake,(-1,1)).
card(tanuki,(2,1)).  card(tanuki,(0,-1)).  card(tanuki,(-1,1)).
card(mouse,(1,0)).  card(mouse,(0,-1)).  card(mouse,(-1,1)).
card(panda,(1,-1)).  card(panda,(0,-1)).  card(panda,(-1,1)).
card(fox,(1,-1)).  card(fox,(1,0)).  card(fox,(1,1)).
```

Logic Program A.2. Five Field Kono

```
role(w).
role(b).

cell(5, 1).  cell(5, 2).  cell(5, 3).  cell(5, 4). cell(5, 5).
cell(4, 1).  cell(4, 2).  cell(4, 3).  cell(4, 4). cell(4, 5).
cell(3, 1).  cell(3, 2).  cell(3, 3).  cell(3, 4). cell(3, 5).
cell(2, 1).  cell(2, 2).  cell(2, 3).  cell(2, 4). cell(2, 5).
cell(1, 1).  cell(1, 2).  cell(1, 3).  cell(1, 4). cell(1, 5).

init(state(1,1,w)).
init(state(1,2,w)).
init(state(1,3,w)).
init(state(1,4,w)).
init(state(1,5,w)).
init(state(2,1,w)).
init(state(2,5,w)).

init(state(5,1,b)).
init(state(5,2,b)).
init(state(5,3,b)).
init(state(5,4,b)).
init(state(5,5,b)).
init(state(4,1,b)).
init(state(4,5,b)).

```

```
state(A, B, e) :- not state(A, B, w), not state(A, B, b), cell(A, B).

not_in_starting_location(w) :- init(state(A, B, b)), not state(A, B, w).
not_in_starting_location(b) :- init(state(A, B, w)), not state(A, B, b).

goal(P, 100) :- not not_in_starting_location(P), role(P).
goal(P, 50) :- not legal(_, _), role(P).
goal(P, 0) :- not_in_starting_location(P).

0 { does(P, A) } 1 :- legal(P, A), not terminal.
:- role(P), not does(P, _), not terminal.
:- does(P, A1), does(P, A2), A1 < A2.

next(control(x)) :- control(o).
next(control(o)) :- control(x).

next(state(A, B, P)) :- role(P), state(A, B, P), not does(P, move(cell(A, B),
 ↪  _)).
next(state(A, B, P)) :- does(P, move(cell(A, B), _)).

next(box(M,P)) :- does(P,mark(M)), init(box(M,b)).
next(box(M,P)) :- init(box(M,P)), P!=b.
next(box(M1,b)) :- does(P,mark(M2)), init(box(M1,b)), M1!=M2.

legal(P, noop) :- role(P), not control(P).

adj(cell(V0, V1), cell(V2, V3)) :- cell(V0, V1), cell(V2, V3), V2 == V0+1, V3 ==
 ↪  V1+1.
adj(cell(V0, V1), cell(V2, V3)) :- cell(V0, V1), cell(V2, V3), V1 == V3+1, V0 ==
 ↪  V2-1.
adj(cell(V2, V3), cell(V0, V1)) :- cell(V0, V1), cell(V2, V3), V1 == V3+1, V0 ==
 ↪  V2-1.
adj(cell(V0, V1), cell(V2, V3)) :- cell(V0, V1), cell(V2, V3), V1 == V3+1, V2 ==
 ↪  V0-1.
legal(V4, move(cell(V0, V1), cell(V2, V3))) :- adj(cell(V0, V1), cell(V2, V3)),
 ↪  state(V0, V1, V4), state(V2, V3, e), control(V4).

terminal :- goal(_, 100).
terminal :- goal(_, 50).
```

Logic Program A.3. Cross-Dot Game

```
role(x).
role(o).
box(1).
box(2).
box(3).
box(4).
next(box(M,P)) :- does(P,mark(M)), box(M,b).
next(box(M,P)) :- box(M,P), P!=b.
next(box(M1,b)) :- does(_,mark(M2)), box(M1,b), M1!=M2.
```

```
next(control(x)) :- control(o).
next(control(o)) :- control(x).
legal(P,mark(M)) :- box(M,b), control(P).
legal(P,noop) :- not control(P), role(P).
legal(P,noop) :- terminal, role(P).
open :- box(_,b).
terminal :- not open.
terminal :- goal(_, 100).
chain_segments(1,M,P) :- role(P), box(M,P).
chain_segments(2,M+1,P) :- chain_segments(1, M, P), box(M+1,P).
longest_chain(M,P) :- chain_segments(M,_,P), not chain_segments(M+1,_,P).
longest_chain(0,P) :- role(P), not chain_segments(_,_,P).
0 { does(P, mark(B)) } 1 :- legal(P, mark(B)), not terminal.
does(P, noop) :- legal(P, noop).
:- role(P), not does(P,_), not terminal.
:- does(P,mark(B1)), does(P,mark(B2)), B1<B2.
goal(P,100) :- longest_chain(2,P).
goal(o,0) :- goal(x, 100).
goal(x,0) :- goal(o, 100).
adj(M,M+1) :- box(M), box(M+1).
adj(M+1,M) :- box(M), box(M+1).
```

B ILASP LEARNING EXAMPLES

B.1 ONITAMA

B.1.1 EXPERIMENT 9.1: ONITAMA RULES

Logic Program B.1. Examples of legal and illegal moves in Onitama

```
#pos({legal(blue,move((cell(1,2),cell(2,3)),rabbit))}, {}, {
in_play(seasnake).
in_play(dog).
in_play(rabbit).
control(blue).
in_hand(red,dog).
in_hand(blue,rabbit).
location(pawn(master,red),cell(5,3)).
location(pawn(student,red),cell(5,1)).
location(pawn(student,red),cell(5,2)).
location(pawn(student,red),cell(5,4)).
location(pawn(student,red),cell(5,5)).
location(pawn(master,blue),cell(1,3)).
location(pawn(student,blue),cell(1,1)).
location(pawn(student,blue),cell(1,2)).
location(pawn(student,blue),cell(1,4)).
location(pawn(student,blue),cell(1,5)).
card(seasnake,(2,0)).  card(seasnake,(0,-1)). card(seasnake,(-1,1)).
card(dog,(-1,-1)).  card(dog,(-1,0)).  card(dog,(-1,1)).
card(rabbit,(1,-1)).  card(rabbit,(-1,1)).  card(rabbit,(2,0)).
}).

#pos({ legal(blue,move((cell(2,3),cell(3,3)),seasnake))
, legal(blue,move((cell(2,3),cell(1,2)),seasnake))
},
{ legal(blue,move((cell(1,5),cell(1,7)),seasnake))
, legal(blue,move((cell(1,5),cell(2,4)),seasnake))
, legal(blue,move((cell(2,3),cell(1,4)),seasnake))
, legal(red,move((cell(4,1),cell(5,2)),rabbit))
}, {
in_play(dog).
in_play(rabbit).
in_play(seasnake).
control(blue).
in_hand(red,rabbit).
in_hand(blue,seasnake).
location(pawn(master,red),cell(3,4)).
location(pawn(student,red),cell(4,1)).
location(pawn(student,red),cell(4,5)).
location(pawn(master,blue),cell(1,3)).
```

```
location(pawn(student,blue),cell(2,1)).
location(pawn(student,blue),cell(2,3)).
location(pawn(student,blue),cell(1,5)).
card(seasnake,(2,0)).  card(seasnake,(0,-1)). card(seasnake,(-1,1)).
card(dog,(-1,-1)).  card(dog,(-1,0)).  card(dog,(-1,1)).
card(rabbit,(1,-1)).  card(rabbit,(-1,1)).  card(rabbit,(2,0)).
}).
#pos(
{ legal(red,move((cell(4,3),cell(2,2)),kirin))
},
{ legal(red,move((cell(3,3),cell(5,3)),kirin))
, legal(red,move((cell(3,2),cell(1,1)),kirin))
, legal(red,move((cell(3,3),cell(2,3)),rat))
}, {
in_play(fox).
in_play(kirin).
in_play(rat).
control(red).
in_hand(red,kirin).
in_hand(blue,rat).
location(pawn(master,red),cell(5,3)).
location(pawn(student,red),cell(4,3)).
location(pawn(student,red),cell(3,3)).
location(pawn(master,blue),cell(3,5)).
location(pawn(student,blue),cell(4,4)).
location(pawn(student,blue),cell(2,2)).
card(fox,(1,-1)).  card(fox,(1,0)).  card(fox,(1,1)).
card(rat,(-1,0)).  card(rat,(0,-1)).  card(rat,(1,1)).
card(kirin,(-1,-2)).  card(kirin,(1,-2)).  card(kirin,(0,2)).
}).

#max_penalty(1000).
```

```
#bias(":- head(pred_1(V0, (V1, V1))).").
#bias(":- not head(pred_3(_, _)), body(location(_, _)).").
#bias(":- not head(pred_4(_, _)), body(naf(location(_, _))).").
#bias(":- not head(pred_3(_, _)), not head(pred_4(_, _)), body(control(_)).").

#bias(":- not head(pred_5(_, _)), body(cell(_, _)).").
#bias(":- head(pred_5((cell(V0, V0), _), _)).").
#bias(":- head(pred_5((_, cell(V0, V0)), _)).").
#bias(":- head(pred_5((cell(_, V0), cell(_, V0)), _)).").
#bias(":- head(pred_5((cell(_, V0), cell(V0, _)), _)).").
#bias(":- head(pred_5((cell(V0, _), cell(_, V0)), _)).").
#bias(":- head(pred_5((cell(V0, _), cell(V0, _)), _)).").
#bias(":- head(pred_5(_, (V0, V0))).").
#bias(":- body(pred_5(_, (V0, V0))).").

#bias(":- head(pred_5(_, _)), body(cell(V0, _)), body(cell(_, V0)).").

#bias(":- not head(pred_2(_, _, _)), body(card(_, _)).").
#bias(":- not head(pred_2(_, _, _)), body(pred_6(_, _, _)).").
```

```
#bias(":- body(card(_, (V0, V0)))").

#bias(":- head(pred_3(_, _)), body(pred_1(_, _)).").
#bias(":- not head(pred_1(_, _)), body(pred_3(_, _)).").
#bias(":- not head(pred_1(_, _)), body(pred_4(_, _)).").
#bias(":- not head(pred_6(_, _, _)), body(math(_, _, _, _)).").
#bias(":- not head(pred_6(_, _, _)), body(pred_5(_, _)).").

#bias(":- head(legal(_, move(V1, V1, _)))").
#bias(":- body(math(V0, V0, _, _)).").
```

B.1.2 EXPERIMENT 10.7: DEFEND PAWNS

```
% Context: C1
location(pawn(student,red),cell(5,1)).
location(pawn(student,red),cell(5,2)).
location(pawn(master,red),cell(5,3)).
location(pawn(student,red),cell(5,4)).
location(pawn(student,red),cell(4,4)).
location(pawn(student,blue),cell(1,1)).
location(pawn(student,blue),cell(1,2)).
location(pawn(master,blue),cell(1,3)).
location(pawn(student,blue),cell(1,4)).
location(pawn(student,blue),cell(1,5)).
control(blue).
in_play(rat).
in_play(crab).
in_play(mantis).
in_hand(red,rat).
in_hand(blue,crab).

% Context: C2
location(pawn(student,red),cell(5,1)).
location(pawn(student,red),cell(5,2)).
location(pawn(master,red),cell(5,3)).
location(pawn(student,red),cell(5,5)).
location(pawn(student,red),cell(4,5)).
location(pawn(student,blue),cell(1,1)).
location(pawn(student,blue),cell(1,2)).
location(pawn(master,blue),cell(1,3)).
location(pawn(student,blue),cell(1,4)).
location(pawn(student,blue),cell(1,5)).
control(blue).
in_play(rat).
in_play(crab).
in_play(mantis).
in_hand(red,rat).
in_hand(blue,crab).

% Context: C3
location(pawn(student,red),cell(5,1)).
location(pawn(master,red),cell(5,3)).
location(pawn(student,red),cell(5,4)).
location(pawn(student,red),cell(4,2)).
location(pawn(student,red),cell(4,4)).
location(pawn(student,blue),cell(2,1)).
location(pawn(student,blue),cell(2,5)).
location(pawn(student,blue),cell(1,1)).
location(pawn(master,blue),cell(1,3)).
location(pawn(student,blue),cell(1,4)).
control(red).
in_play(crab).
in_play(rat).
in_play(mantis).
in_hand(red,crab).
in_hand(blue,rat).

% Context: C4
location(pawn(student,red),cell(5,1)).
location(pawn(master,red),cell(5,3)).
location(pawn(student,red),cell(5,4)).
location(pawn(student,red),cell(4,2)).
location(pawn(student,red),cell(4,4)).
location(pawn(master,blue),cell(2,2)).
location(pawn(student,blue),cell(2,5)).
location(pawn(student,blue),cell(1,1)).
location(pawn(student,blue),cell(1,2)).
location(pawn(student,blue),cell(1,4)).
control(red).
in_play(crab).
in_play(rat).
in_play(mantis).
in_hand(red,crab).
in_hand(blue,rat).

% Context: C5
location(pawn(master,red),cell(5,3)).
location(pawn(student,red),cell(5,4)).
location(pawn(student,red),cell(4,1)).
location(pawn(student,red),cell(4,2)).
location(pawn(student,red),cell(4,4)).
location(pawn(student,blue),cell(2,1)).
location(pawn(student,blue),cell(2,4)).
location(pawn(student,blue),cell(1,1)).
location(pawn(master,blue),cell(1,3)).
location(pawn(student,blue),cell(1,4)).
control(red).
in_play(mantis).
in_play(crab).
```

```
in_play(rat).
in_hand(red,mantis).
in_hand(blue,crab).

% Context: C6
location(pawn(master,red),cell(5,3)).
location(pawn(student,red),cell(5,4)).
location(pawn(student,red),cell(4,1)).
location(pawn(student,red),cell(4,2)).
location(pawn(student,red),cell(4,4)).
location(pawn(student,blue),cell(2,1)).
location(pawn(master,blue),cell(2,3)).
location(pawn(student,blue),cell(2,5)).
location(pawn(student,blue),cell(1,1)).
location(pawn(student,blue),cell(1,4)).
control(red).
in_play(mantis).
in_play(crab).
in_play(rat).
in_hand(red,mantis).
in_hand(blue,crab).
```

Logic Program B.2. Deep Ordering Translation for the Defend Task

```
location(pawn(master,red),cell(5,3)).
location(pawn(student,blue),cell(2,1)).
location(pawn(student,blue),cell(2,4)).
location(pawn(student,blue),cell(1,1)).
location(pawn(master,blue),cell(1,3)).
location(pawn(student,blue),cell(1,4)).
in_hand(blue,crab).
in_hand(red,rat).
control(blue).
location(pawn(student,red),cell(5,4)).
location(pawn(student,red),cell(4,1)).
location(pawn(student,red),cell(4,4)).
location(pawn(student,red),cell(3,1)).
does(blue,move(cell(2,4),cell(2,2),crab)).
does(red,noop).
}).
#pos(ex_a_0_4, {}, {}, {
location(pawn(master,red),cell(5,3)).
location(pawn(student,blue),cell(2,1)).
location(pawn(student,blue),cell(2,4)).
location(pawn(student,blue),cell(1,1)).
location(pawn(master,blue),cell(1,3)).
location(pawn(student,blue),cell(1,4)).
in_hand(blue,crab).
in_hand(red,rat).
control(blue).
location(pawn(student,red),cell(5,4)).
location(pawn(student,red),cell(4,1)).
location(pawn(student,red),cell(4,4)).
location(pawn(student,red),cell(3,1)).
does(blue,move(cell(2,4),cell(3,4),crab)).
does(red,noop).
}).
#pos(ex_a_0_5, {}, {}, {
location(pawn(master,red),cell(5,3)).
location(pawn(student,blue),cell(2,1)).
location(pawn(student,blue),cell(2,4)).
location(pawn(student,blue),cell(1,1)).
location(pawn(master,blue),cell(1,3)).
```

```
location(pawn(student,blue),cell(1,4)).
in_hand(blue,crab).
in_hand(red,rat).
control(blue).
location(pawn(student,red),cell(5,4)).
location(pawn(student,red),cell(4,1)).
location(pawn(student,red),cell(4,4)).
location(pawn(student,red),cell(3,1)).
does(blue,move(cell(1,3),cell(1,5),crab)).
does(red,noop).
}).
#pos(ex_a_0_6, {}, {}, {
location(pawn(master,red),cell(5,3)).
location(pawn(student,blue),cell(2,1)).
location(pawn(student,blue),cell(2,4)).
location(pawn(student,blue),cell(1,1)).
location(pawn(master,blue),cell(1,3)).
location(pawn(student,blue),cell(1,4)).
in_hand(blue,crab).
in_hand(red,rat).
control(blue).
location(pawn(student,red),cell(5,4)).
location(pawn(student,red),cell(4,1)).
location(pawn(student,red),cell(4,4)).
location(pawn(student,red),cell(3,1)).
does(blue,move(cell(2,1),cell(2,3),crab)).
does(red,noop).
}).
#pos(ex_a_0_7, {}, {}, {
location(pawn(master,red),cell(5,3)).
location(pawn(student,blue),cell(2,1)).
location(pawn(student,blue),cell(2,4)).
location(pawn(student,blue),cell(1,1)).
location(pawn(master,blue),cell(1,3)).
location(pawn(student,blue),cell(1,4)).
in_hand(blue,crab).
in_hand(red,rat).
control(blue).
location(pawn(student,red),cell(5,4)).
location(pawn(student,red),cell(4,1)).
location(pawn(student,red),cell(4,4)).
location(pawn(student,red),cell(3,1)).
does(blue,move(cell(1,3),cell(2,3),crab)).
does(red,noop).
}).
#pos(ex_a_1_1, {}, {}, {
location(pawn(master,red),cell(5,3)).
location(pawn(student,blue),cell(2,1)).
location(pawn(student,blue),cell(2,4)).
location(pawn(student,blue),cell(1,1)).
location(pawn(master,blue),cell(1,3)).
```

```
location(pawn(student,blue),cell(1,4)).
in_hand(blue,crab).
in_hand(red,rat).
control(blue).
location(pawn(student,red),cell(4,1)).
location(pawn(student,red),cell(4,2)).
location(pawn(student,red),cell(4,4)).
location(pawn(student,red),cell(4,5)).
does(blue,move(cell(1,4),cell(1,2),crab)).
does(red,noop).
}).
#pos(ex_a_1_2, {}, {}, {
location(pawn(master,red),cell(5,3)).
location(pawn(student,blue),cell(2,1)).
location(pawn(student,blue),cell(2,4)).
location(pawn(student,blue),cell(1,1)).
location(pawn(master,blue),cell(1,3)).
location(pawn(student,blue),cell(1,4)).
in_hand(blue,crab).
in_hand(red,rat).
control(blue).
location(pawn(student,red),cell(4,1)).
location(pawn(student,red),cell(4,2)).
location(pawn(student,red),cell(4,4)).
location(pawn(student,red),cell(4,5)).
does(blue,move(cell(2,1),cell(3,1),crab)).
does(red,noop).
}).
#pos(ex_a_1_3, {}, {}, {
location(pawn(master,red),cell(5,3)).
location(pawn(student,blue),cell(2,1)).
location(pawn(student,blue),cell(2,4)).
location(pawn(student,blue),cell(1,1)).
location(pawn(master,blue),cell(1,3)).
location(pawn(student,blue),cell(1,4)).
in_hand(blue,crab).
in_hand(red,rat).
control(blue).
location(pawn(student,red),cell(4,1)).
location(pawn(student,red),cell(4,2)).
location(pawn(student,red),cell(4,4)).
location(pawn(student,red),cell(4,5)).
does(blue,move(cell(2,4),cell(2,2),crab)).
does(red,noop).
}).
#pos(ex_a_1_4, {}, {}, {
location(pawn(master,red),cell(5,3)).
location(pawn(student,blue),cell(2,1)).
location(pawn(student,blue),cell(2,4)).
location(pawn(student,blue),cell(1,1)).
location(pawn(master,blue),cell(1,3)).
```

```
location(pawn(student,blue),cell(1,4)).
in_hand(blue,crab).
in_hand(red,rat).
control(blue).
location(pawn(student,red),cell(4,1)).
location(pawn(student,red),cell(4,2)).
location(pawn(student,red),cell(4,4)).
location(pawn(student,red),cell(4,5)).
does(blue,move(cell(2,4),cell(3,4),crab)).
does(red,noop).
}).
#pos(ex_a_1_5, {}, {}, {
location(pawn(master,red),cell(5,3)).
location(pawn(student,blue),cell(2,1)).
location(pawn(student,blue),cell(2,4)).
location(pawn(student,blue),cell(1,1)).
location(pawn(master,blue),cell(1,3)).
location(pawn(student,blue),cell(1,4)).
in_hand(blue,crab).
in_hand(red,rat).
control(blue).
location(pawn(student,red),cell(4,1)).
location(pawn(student,red),cell(4,2)).
location(pawn(student,red),cell(4,4)).
location(pawn(student,red),cell(4,5)).
does(blue,move(cell(1,3),cell(1,5),crab)).
does(red,noop).
}).
#pos(ex_a_1_6, {}, {}, {
location(pawn(master,red),cell(5,3)).
location(pawn(student,blue),cell(2,1)).
location(pawn(student,blue),cell(2,4)).
location(pawn(student,blue),cell(1,1)).
location(pawn(master,blue),cell(1,3)).
location(pawn(student,blue),cell(1,4)).
in_hand(blue,crab).
in_hand(red,rat).
control(blue).
location(pawn(student,red),cell(4,1)).
location(pawn(student,red),cell(4,2)).
location(pawn(student,red),cell(4,4)).
location(pawn(student,red),cell(4,5)).
does(blue,move(cell(2,1),cell(2,3),crab)).
does(red,noop).
}).
#pos(ex_a_1_7, {}, {}, {
location(pawn(master,red),cell(5,3)).
location(pawn(student,blue),cell(2,1)).
location(pawn(student,blue),cell(2,4)).
location(pawn(student,blue),cell(1,1)).
location(pawn(master,blue),cell(1,3)).
```

```
location(pawn(student,blue),cell(1,4)).
in_hand(blue,crab).
in_hand(red,rat).
control(blue).
location(pawn(student,red),cell(4,1)).
location(pawn(student,red),cell(4,2)).
location(pawn(student,red),cell(4,4)).
location(pawn(student,red),cell(4,5)).
does(blue,move(cell(1,3),cell(2,3),crab)).
does(red,noop).
}).
#brave_ordering(ord_a_1_ex_a_0_1_ex_a_1_1, ex_a_0_1, ex_a_1_1).
#brave_ordering(ord_a_1_ex_a_0_1_ex_a_1_2, ex_a_0_1, ex_a_1_2).
#brave_ordering(ord_a_1_ex_a_0_1_ex_a_1_3, ex_a_0_1, ex_a_1_3).
#brave_ordering(ord_a_1_ex_a_0_1_ex_a_1_4, ex_a_0_1, ex_a_1_4).
#brave_ordering(ord_a_1_ex_a_0_1_ex_a_1_5, ex_a_0_1, ex_a_1_5).
#brave_ordering(ord_a_1_ex_a_0_1_ex_a_1_6, ex_a_0_1, ex_a_1_6).
#brave_ordering(ord_a_1_ex_a_0_1_ex_a_1_7, ex_a_0_1, ex_a_1_7).
#brave_ordering(ord_a_1_ex_a_0_2_ex_a_1_1, ex_a_0_2, ex_a_1_1).
#brave_ordering(ord_a_1_ex_a_0_2_ex_a_1_2, ex_a_0_2, ex_a_1_2).
#brave_ordering(ord_a_1_ex_a_0_2_ex_a_1_3, ex_a_0_2, ex_a_1_3).
#brave_ordering(ord_a_1_ex_a_0_2_ex_a_1_4, ex_a_0_2, ex_a_1_4).
#brave_ordering(ord_a_1_ex_a_0_2_ex_a_1_5, ex_a_0_2, ex_a_1_5).
#brave_ordering(ord_a_1_ex_a_0_2_ex_a_1_6, ex_a_0_2, ex_a_1_6).
#brave_ordering(ord_a_1_ex_a_0_2_ex_a_1_7, ex_a_0_2, ex_a_1_7).
#brave_ordering(ord_a_1_ex_a_0_3_ex_a_1_1, ex_a_0_3, ex_a_1_1).
#brave_ordering(ord_a_1_ex_a_0_3_ex_a_1_2, ex_a_0_3, ex_a_1_2).
#brave_ordering(ord_a_1_ex_a_0_3_ex_a_1_3, ex_a_0_3, ex_a_1_3).
#brave_ordering(ord_a_1_ex_a_0_3_ex_a_1_4, ex_a_0_3, ex_a_1_4).
#brave_ordering(ord_a_1_ex_a_0_3_ex_a_1_5, ex_a_0_3, ex_a_1_5).
#brave_ordering(ord_a_1_ex_a_0_3_ex_a_1_6, ex_a_0_3, ex_a_1_6).
#brave_ordering(ord_a_1_ex_a_0_3_ex_a_1_7, ex_a_0_3, ex_a_1_7).
#brave_ordering(ord_a_1_ex_a_0_4_ex_a_1_1, ex_a_0_4, ex_a_1_1).
#brave_ordering(ord_a_1_ex_a_0_4_ex_a_1_2, ex_a_0_4, ex_a_1_2).
#brave_ordering(ord_a_1_ex_a_0_4_ex_a_1_3, ex_a_0_4, ex_a_1_3).
#brave_ordering(ord_a_1_ex_a_0_4_ex_a_1_4, ex_a_0_4, ex_a_1_4).
#brave_ordering(ord_a_1_ex_a_0_4_ex_a_1_5, ex_a_0_4, ex_a_1_5).
#brave_ordering(ord_a_1_ex_a_0_4_ex_a_1_6, ex_a_0_4, ex_a_1_6).
#brave_ordering(ord_a_1_ex_a_0_4_ex_a_1_7, ex_a_0_4, ex_a_1_7).
#brave_ordering(ord_a_1_ex_a_0_5_ex_a_1_1, ex_a_0_5, ex_a_1_1).
#brave_ordering(ord_a_1_ex_a_0_5_ex_a_1_2, ex_a_0_5, ex_a_1_2).
#brave_ordering(ord_a_1_ex_a_0_5_ex_a_1_3, ex_a_0_5, ex_a_1_3).
#brave_ordering(ord_a_1_ex_a_0_5_ex_a_1_4, ex_a_0_5, ex_a_1_4).
#brave_ordering(ord_a_1_ex_a_0_5_ex_a_1_5, ex_a_0_5, ex_a_1_5).
#brave_ordering(ord_a_1_ex_a_0_5_ex_a_1_6, ex_a_0_5, ex_a_1_6).
#brave_ordering(ord_a_1_ex_a_0_5_ex_a_1_7, ex_a_0_5, ex_a_1_7).
#brave_ordering(ord_a_1_ex_a_0_6_ex_a_1_1, ex_a_0_6, ex_a_1_1).
#brave_ordering(ord_a_1_ex_a_0_6_ex_a_1_2, ex_a_0_6, ex_a_1_2).
#brave_ordering(ord_a_1_ex_a_0_6_ex_a_1_3, ex_a_0_6, ex_a_1_3).
#brave_ordering(ord_a_1_ex_a_0_6_ex_a_1_4, ex_a_0_6, ex_a_1_4).
#brave_ordering(ord_a_1_ex_a_0_6_ex_a_1_5, ex_a_0_6, ex_a_1_5).
```

```
#brave_ordering(ord_a_1_ex_a_0_6_ex_a_1_6, ex_a_0_6, ex_a_1_6).
#brave_ordering(ord_a_1_ex_a_0_6_ex_a_1_7, ex_a_0_6, ex_a_1_7).
#brave_ordering(ord_a_1_ex_a_0_7_ex_a_1_1, ex_a_0_7, ex_a_1_1).
#brave_ordering(ord_a_1_ex_a_0_7_ex_a_1_2, ex_a_0_7, ex_a_1_2).
#brave_ordering(ord_a_1_ex_a_0_7_ex_a_1_3, ex_a_0_7, ex_a_1_3).
#brave_ordering(ord_a_1_ex_a_0_7_ex_a_1_4, ex_a_0_7, ex_a_1_4).
#brave_ordering(ord_a_1_ex_a_0_7_ex_a_1_5, ex_a_0_7, ex_a_1_5).
#brave_ordering(ord_a_1_ex_a_0_7_ex_a_1_6, ex_a_0_7, ex_a_1_6).
#brave_ordering(ord_a_1_ex_a_0_7_ex_a_1_7, ex_a_0_7, ex_a_1_7).
#inject("r_o_o_t(ex_a_0,chosen).
r_o_o_t(ex_a_1,other).
c_h_i_l_d(ex_a_0,ex_a_0_1).
c_h_i_l_d(ex_a_0,ex_a_0_2).
c_h_i_l_d(ex_a_0,ex_a_0_3).
c_h_i_l_d(ex_a_0,ex_a_0_4).
c_h_i_l_d(ex_a_0,ex_a_0_5).
c_h_i_l_d(ex_a_0,ex_a_0_6).
c_h_i_l_d(ex_a_0,ex_a_0_7).
c_h_i_l_d(ex_a_1,ex_a_1_1).
c_h_i_l_d(ex_a_1,ex_a_1_2).
c_h_i_l_d(ex_a_1,ex_a_1_3).
c_h_i_l_d(ex_a_1,ex_a_1_4).
c_h_i_l_d(ex_a_1,ex_a_1_5).
c_h_i_l_d(ex_a_1,ex_a_1_6).
c_h_i_l_d(ex_a_1,ex_a_1_7).
:- c_h_i_l_d(P1,C), c_h_i_l_d(P2,C), P1<P2.
example_active(EX_ID,forall) :- r_o_o_t(EX_ID,chosen).
example_active(EX_ID,exists) :- r_o_o_t(EX_ID,other).
1 {example_active(Child,forall) : c_h_i_l_d(Parent,Child)} 1 :-
 ↪  example_active(Parent,exists), c_h_i_l_d(Parent,_).
example_active(Child,exists) :- c_h_i_l_d(Parent,Child),
 ↪  example_active(Parent,forall).
example_active(ORD_ID) :- o_r_d(EX_ID_1,EX_ID_2,ORD_ID),
 ↪  example_active(EX_ID_1,_), example_active(EX_ID_2,_).
o_r_d(ex_a_0_1,ex_a_1_1,ord_a_1_ex_a_0_1_ex_a_1_1).
o_r_d(ex_a_0_1,ex_a_1_2,ord_a_1_ex_a_0_1_ex_a_1_2).
o_r_d(ex_a_0_1,ex_a_1_3,ord_a_1_ex_a_0_1_ex_a_1_3).
o_r_d(ex_a_0_1,ex_a_1_4,ord_a_1_ex_a_0_1_ex_a_1_4).
o_r_d(ex_a_0_1,ex_a_1_5,ord_a_1_ex_a_0_1_ex_a_1_5).
o_r_d(ex_a_0_1,ex_a_1_6,ord_a_1_ex_a_0_1_ex_a_1_6).
o_r_d(ex_a_0_1,ex_a_1_7,ord_a_1_ex_a_0_1_ex_a_1_7).
o_r_d(ex_a_0_2,ex_a_1_1,ord_a_1_ex_a_0_2_ex_a_1_1).
o_r_d(ex_a_0_2,ex_a_1_2,ord_a_1_ex_a_0_2_ex_a_1_2).
o_r_d(ex_a_0_2,ex_a_1_3,ord_a_1_ex_a_0_2_ex_a_1_3).
o_r_d(ex_a_0_2,ex_a_1_4,ord_a_1_ex_a_0_2_ex_a_1_4).
o_r_d(ex_a_0_2,ex_a_1_5,ord_a_1_ex_a_0_2_ex_a_1_5).
o_r_d(ex_a_0_2,ex_a_1_6,ord_a_1_ex_a_0_2_ex_a_1_6).
o_r_d(ex_a_0_2,ex_a_1_7,ord_a_1_ex_a_0_2_ex_a_1_7).
o_r_d(ex_a_0_3,ex_a_1_1,ord_a_1_ex_a_0_3_ex_a_1_1).
o_r_d(ex_a_0_3,ex_a_1_2,ord_a_1_ex_a_0_3_ex_a_1_2).
o_r_d(ex_a_0_3,ex_a_1_3,ord_a_1_ex_a_0_3_ex_a_1_3).
```

```
o_r_d(ex_a_0_3,ex_a_1_4,ord_a_1_ex_a_0_3_ex_a_1_4).
o_r_d(ex_a_0_3,ex_a_1_5,ord_a_1_ex_a_0_3_ex_a_1_5).
o_r_d(ex_a_0_3,ex_a_1_6,ord_a_1_ex_a_0_3_ex_a_1_6).
o_r_d(ex_a_0_3,ex_a_1_7,ord_a_1_ex_a_0_3_ex_a_1_7).
o_r_d(ex_a_0_4,ex_a_1_1,ord_a_1_ex_a_0_4_ex_a_1_1).
o_r_d(ex_a_0_4,ex_a_1_2,ord_a_1_ex_a_0_4_ex_a_1_2).
o_r_d(ex_a_0_4,ex_a_1_3,ord_a_1_ex_a_0_4_ex_a_1_3).
o_r_d(ex_a_0_4,ex_a_1_4,ord_a_1_ex_a_0_4_ex_a_1_4).
o_r_d(ex_a_0_4,ex_a_1_5,ord_a_1_ex_a_0_4_ex_a_1_5).
o_r_d(ex_a_0_4,ex_a_1_6,ord_a_1_ex_a_0_4_ex_a_1_6).
o_r_d(ex_a_0_4,ex_a_1_7,ord_a_1_ex_a_0_4_ex_a_1_7).
o_r_d(ex_a_0_5,ex_a_1_1,ord_a_1_ex_a_0_5_ex_a_1_1).
o_r_d(ex_a_0_5,ex_a_1_2,ord_a_1_ex_a_0_5_ex_a_1_2).
o_r_d(ex_a_0_5,ex_a_1_3,ord_a_1_ex_a_0_5_ex_a_1_3).
o_r_d(ex_a_0_5,ex_a_1_4,ord_a_1_ex_a_0_5_ex_a_1_4).
o_r_d(ex_a_0_5,ex_a_1_5,ord_a_1_ex_a_0_5_ex_a_1_5).
o_r_d(ex_a_0_5,ex_a_1_6,ord_a_1_ex_a_0_5_ex_a_1_6).
o_r_d(ex_a_0_5,ex_a_1_7,ord_a_1_ex_a_0_5_ex_a_1_7).
o_r_d(ex_a_0_6,ex_a_1_1,ord_a_1_ex_a_0_6_ex_a_1_1).
o_r_d(ex_a_0_6,ex_a_1_2,ord_a_1_ex_a_0_6_ex_a_1_2).
o_r_d(ex_a_0_6,ex_a_1_3,ord_a_1_ex_a_0_6_ex_a_1_3).
o_r_d(ex_a_0_6,ex_a_1_4,ord_a_1_ex_a_0_6_ex_a_1_4).
o_r_d(ex_a_0_6,ex_a_1_5,ord_a_1_ex_a_0_6_ex_a_1_5).
o_r_d(ex_a_0_6,ex_a_1_6,ord_a_1_ex_a_0_6_ex_a_1_6).
o_r_d(ex_a_0_6,ex_a_1_7,ord_a_1_ex_a_0_6_ex_a_1_7).
o_r_d(ex_a_0_7,ex_a_1_1,ord_a_1_ex_a_0_7_ex_a_1_1).
o_r_d(ex_a_0_7,ex_a_1_2,ord_a_1_ex_a_0_7_ex_a_1_2).
o_r_d(ex_a_0_7,ex_a_1_3,ord_a_1_ex_a_0_7_ex_a_1_3).
o_r_d(ex_a_0_7,ex_a_1_4,ord_a_1_ex_a_0_7_ex_a_1_4).
o_r_d(ex_a_0_7,ex_a_1_5,ord_a_1_ex_a_0_7_ex_a_1_5).
o_r_d(ex_a_0_7,ex_a_1_6,ord_a_1_ex_a_0_7_ex_a_1_6).
o_r_d(ex_a_0_7,ex_a_1_7,ord_a_1_ex_a_0_7_ex_a_1_7).
example_active(ex_d_0).
example_active(ex_d_1).
example_active(ex_c_0).
example_active(ex_c_1).
example_active(ex_b_0).
example_active(ex_b_1).
example_active(ex_a_0).
example_active(ex_a_1).
example_active(ex_a_0_1).
example_active(ex_a_0_2).
example_active(ex_a_0_3).
example_active(ex_a_0_4).
example_active(ex_a_0_5).
example_active(ex_a_0_6).
example_active(ex_a_0_7).
example_active(ex_a_1_1).
example_active(ex_a_1_2).
example_active(ex_a_1_3).
example_active(ex_a_1_4).
```

```
example_active(ex_a_1_5).
example_active(ex_a_1_6).
example_active(ex_a_1_7).").
```

B.2 FIVE FIELD KONO

B.2.1 EXPERIMENT 9.2: FIVE FIELD KONO RULES

Logic Program B.3. Examples of legal and illegal moves in Five Field Kono

```
#pos(init,
{ legal(w, move(cell(1,2), cell(2,3)))
, legal(b, noop)
, legal(w, move(cell(2,5), cell(3,4)))
},
{ legal(w, noop)
, legal(w, move(cell(1,2), cell(2,2)))
, legal(w, move(cell(1,2), cell(1,1)))
},
{
control(w).

state(1,1,w).
state(1,2,w).
state(1,3,w).
state(1,4,w).
state(1,5,w).
state(2,1,w).
state(2,5,w).

state(5,1,b).
state(5,2,b).
state(5,3,b).
state(5,4,b).
state(5,5,b).
state(4,1,b).
state(4,5,b).
}).

#pos(mid,
{ legal(b, move(cell(5,1), cell(4,2)))
, legal(w, noop)
, legal(b, move(cell(4,3), cell(5,4)))
},
{ legal(b, noop)
, legal(b, move(cell(4,3), cell(3,4)))
, legal(b, move(cell(4,2), cell(3,1)))
, legal(b, move(cell(4,2), cell(3,3)))
, legal(w, move(cell(1,2), cell(2,3)))
, legal(w, move(cell(2,3), cell(3,2)))
},
{
control(b).

state(1,1,w).
state(2,3,w).
state(1,3,w).
```

```
state(1,4,w).
state(1,5,w).
state(2,1,w).
state(3,4,w).

state(5,1,b).
state(5,2,b).
state(5,3,b).
state(4,3,b).
state(5,5,b).
state(4,1,b).
state(4,5,b).
}).

3 ~ pred(cell(V0, V1), cell(V3, V4), V2) :- state(V0, V1, V2), state(V3, V4, e).
3 ~ pred(cell(V0, V1), cell(V4, V3), V2) :- state(V0, V1, V2), state(V3, V4, e).
3 ~ pred(cell(V0, V3), cell(V1, V4), V2) :- state(V0, V1, V2), state(V3, V4, e).
3 ~ pred(cell(V0, V3), cell(V4, V1), V2) :- state(V0, V1, V2), state(V3, V4, e).
3 ~ pred(cell(V0, V4), cell(V1, V3), V2) :- state(V0, V1, V2), state(V3, V4, e).
3 ~ pred(cell(V0, V4), cell(V3, V1), V2) :- state(V0, V1, V2), state(V3, V4, e).
3 ~ pred(cell(V1, V0), cell(V3, V4), V2) :- state(V0, V1, V2), state(V3, V4, e).
3 ~ pred(cell(V1, V0), cell(V4, V3), V2) :- state(V0, V1, V2), state(V3, V4, e).
3 ~ pred(cell(V1, V3), cell(V0, V4), V2) :- state(V0, V1, V2), state(V3, V4, e).
3 ~ pred(cell(V1, V3), cell(V4, V0), V2) :- state(V0, V1, V2), state(V3, V4, e).
3 ~ pred(cell(V1, V4), cell(V0, V3), V2) :- state(V0, V1, V2), state(V3, V4, e).
3 ~ pred(cell(V1, V4), cell(V3, V0), V2) :- state(V0, V1, V2), state(V3, V4, e).
3 ~ pred(cell(V3, V0), cell(V1, V4), V2) :- state(V0, V1, V2), state(V3, V4, e).
3 ~ pred(cell(V3, V0), cell(V4, V1), V2) :- state(V0, V1, V2), state(V3, V4, e).
3 ~ pred(cell(V3, V1), cell(V0, V4), V2) :- state(V0, V1, V2), state(V3, V4, e).
3 ~ pred(cell(V3, V1), cell(V4, V0), V2) :- state(V0, V1, V2), state(V3, V4, e).
3 ~ pred(cell(V3, V4), cell(V0, V1), V2) :- state(V0, V1, V2), state(V3, V4, e).
3 ~ pred(cell(V3, V4), cell(V1, V0), V2) :- state(V0, V1, V2), state(V3, V4, e).
3 ~ pred(cell(V4, V0), cell(V1, V3), V2) :- state(V0, V1, V2), state(V3, V4, e).
3 ~ pred(cell(V4, V0), cell(V3, V1), V2) :- state(V0, V1, V2), state(V3, V4, e).
3 ~ pred(cell(V4, V1), cell(V0, V3), V2) :- state(V0, V1, V2), state(V3, V4, e).
3 ~ pred(cell(V4, V1), cell(V3, V0), V2) :- state(V0, V1, V2), state(V3, V4, e).
3 ~ pred(cell(V4, V3), cell(V0, V1), V2) :- state(V0, V1, V2), state(V3, V4, e).
3 ~ pred(cell(V4, V3), cell(V1, V0), V2) :- state(V0, V1, V2), state(V3, V4, e).
5 ~ adj(cell(V0, V1), cell(V2, V3)) :- cell(V0, V1), cell(V2, V3), V0 == V2+1, V1 == V3+1.
5 ~ adj(cell(V2, V3), cell(V0, V1)) :- cell(V0, V1), cell(V2, V3), V0 == V2+1, V1 == V3+1.
5 ~ adj(cell(V0, V1), cell(V2, V3)) :- cell(V0, V1), cell(V2, V3), V0 == V2+1, V3 == V1+1.
5 ~ adj(cell(V2, V3), cell(V0, V1)) :- cell(V0, V1), cell(V2, V3), V0 == V2+1, V3 == V1+1.
5 ~ adj(cell(V0, V1), cell(V2, V3)) :- cell(V0, V1), cell(V2, V3), V1 == V3+1, V2 == V0+1.
5 ~ adj(cell(V2, V3), cell(V0, V1)) :- cell(V0, V1), cell(V2, V3), V1 == V3+1, V2 == V0+1.
5 ~ adj(cell(V0, V1), cell(V2, V3)) :- cell(V0, V1), cell(V2, V3), V2 == V0+1, V3 == V1+1.
5 ~ adj(cell(V2, V3), cell(V0, V1)) :- cell(V0, V1), cell(V2, V3), V2 == V0+1, V3 == V1+1.
5 ~ adj(cell(V0, V1), cell(V2, V3)) :- cell(V0, V1), cell(V2, V3), V0 == V2+1, V1 == V3-1.
5 ~ adj(cell(V2, V3), cell(V0, V1)) :- cell(V0, V1), cell(V2, V3), V0 == V2+1, V1 == V3-1.
5 ~ adj(cell(V0, V1), cell(V2, V3)) :- cell(V0, V1), cell(V2, V3), V0 == V2+1, V3 == V1-1.
5 ~ adj(cell(V2, V3), cell(V0, V1)) :- cell(V0, V1), cell(V2, V3), V0 == V2+1, V3 == V1-1.
5 ~ adj(cell(V0, V1), cell(V2, V3)) :- cell(V0, V1), cell(V2, V3), V1 == V3+1, V0 == V2-1.
5 ~ adj(cell(V2, V3), cell(V0, V1)) :- cell(V0, V1), cell(V2, V3), V1 == V3+1, V0 == V2-1.
5 ~ adj(cell(V0, V1), cell(V2, V3)) :- cell(V0, V1), cell(V2, V3), V1 == V3+1, V2 == V0-1.
5 ~ adj(cell(V2, V3), cell(V0, V1)) :- cell(V0, V1), cell(V2, V3), V1 == V3+1, V2 == V0-1.
5 ~ adj(cell(V0, V1), cell(V2, V3)) :- cell(V0, V1), cell(V2, V3), V2 == V0+1, V1 == V3-1.
5 ~ adj(cell(V2, V3), cell(V0, V1)) :- cell(V0, V1), cell(V2, V3), V2 == V0+1, V1 == V3-1.
5 ~ adj(cell(V0, V1), cell(V2, V3)) :- cell(V0, V1), cell(V2, V3), V2 == V0+1, V3 == V1-1.
5 ~ adj(cell(V2, V3), cell(V0, V1)) :- cell(V0, V1), cell(V2, V3), V2 == V0+1, V3 == V1-1.
5 ~ adj(cell(V0, V1), cell(V2, V3)) :- cell(V0, V1), cell(V2, V3), V3 == V1+1, V0 == V2-1.
```

```
5 ~ adj(cell(V2, V3), cell(V0, V1)) :- cell(V0, V1), cell(V2, V3), V3 == V1+1, V0 == V2-1.
5 ~ adj(cell(V0, V1), cell(V2, V3)) :- cell(V0, V1), cell(V2, V3), V3 == V1+1, V2 == V0-1.
5 ~ adj(cell(V2, V3), cell(V0, V1)) :- cell(V0, V1), cell(V2, V3), V3 == V1+1, V2 == V0-1.
5 ~ adj(cell(V0, V1), cell(V2, V3)) :- cell(V0, V1), cell(V2, V3), V0 == V2-1, V1 == V3-1.
5 ~ adj(cell(V2, V3), cell(V0, V1)) :- cell(V0, V1), cell(V2, V3), V0 == V2-1, V1 == V3-1.
5 ~ adj(cell(V0, V1), cell(V2, V3)) :- cell(V0, V1), cell(V2, V3), V0 == V2-1, V3 == V1-1.
5 ~ adj(cell(V2, V3), cell(V0, V1)) :- cell(V0, V1), cell(V2, V3), V0 == V2-1, V3 == V1-1.
5 ~ adj(cell(V0, V1), cell(V2, V3)) :- cell(V0, V1), cell(V2, V3), V1 == V3-1, V2 == V0-1.
5 ~ adj(cell(V2, V3), cell(V0, V1)) :- cell(V0, V1), cell(V2, V3), V1 == V3-1, V2 == V0-1.
5 ~ adj(cell(V0, V1), cell(V2, V3)) :- cell(V0, V1), cell(V2, V3), V2 == V0-1, V3 == V1-1.
5 ~ adj(cell(V2, V3), cell(V0, V1)) :- cell(V0, V1), cell(V2, V3), V2 == V0-1, V3 == V1-1.
#max_penalty(50).
```

B.3 CROSS-DOT

B.3.1 Experiment 9.3: CROSS-DOT RULES

Logic Program B.4. Examples of legal and illegal moves in Cross-Dot

```
#pos({legal(x, mark(1)), legal(o, noop)}, {legal(x, noop), legal(o, mark(1))}, {
  box(1, b).
  box(2, b).
  box(3, b).
  box(4, b).
  control(x).
}).
#pos({legal(x, noop), legal(o, mark(1))}, {legal(o, noop), legal(o, mark(2))}, {
  box(1, b).
  box(2, x).
  box(3, b).
  box(4, b).
  control(o).
}).
#pos({legal(o, noop), legal(x, mark(3))}, {legal(x, mark(1)), legal(x, mark(2))},
 ↪ {
  box(1, o).
  box(2, x).
  box(3, b).
  box(4, b).
  control(x).
}).

#modeh(legal(var(role), mark(var(box)))).
#modeh(legal(var(role), noop)).
#modeb(box(var(box), const(state))).
#modeb(control(var(role))).
#modeb(role(var(role))).

#constant(state, x).
#constant(state, o).
#constant(state, b).

#maxv(2).
```

BIBLIOGRAPHY

Benthem, Johan van (2011). 'Logic Games: From Tools to Models of Interaction'. In: *Proof, Computation and Agency: Logic at the Crossroads*. Ed. by Johan van Benthem, Amitabha Gupta and Rohit Parikh. Dordrecht: Springer Netherlands, pp. 183–216. ISBN: 978-94-007-0080-2. DOI: `10.1007/978-94-007-0080-2_11`. URL: `https://doi.org/10.1007/978-94-007-0080-2{\_}11`.

Blockeel, Hendrik and Luc De Raedt (1998). 'Top-down induction of first-order logical decision trees'. In: *Artificial Intelligence* 101.1-2, pp. 285–297. ISSN: 00043702. DOI: `10.1016/S0004-3702(98)00034-4`. URL: `http://linkinghub.elsevier.com/retrieve/pii/S0004370298000344`.

Buccafurri, Francesco, Nicola Leone and Pasquale Rullo (1997). 'Strong and weak constraints in disjunctive datalog'. In: *Logic Programming And Nonmonotonic Reasoning*. Vol. 1265, pp. 2–17. ISBN: 3-540-63255-7. DOI: `10.1007/3-540-63255-7_2`. URL: `http://dx.doi.org/10.1007/3-540-63255-7{\_}2`.

Calimeri, Francesco et al. (2013). 'ASP-Core-2 Input Language Format'. In: URL: `https://www.mat.unical.it/aspcomp2013/files/ASP-CORE-2.03c.pdf`.

Cerexhe, Timothy, Orkunt Sabuncu and Michael Thielscher (2013). 'Evaluating Answer Set Clause Learning for General Game Playing'. In: *Logic Programming and Nonmonotonic Reasoning*. Ed. by Pedro Cabalar and Tran Cao Son. Berlin, Heidelberg: Springer Berlin Heidelberg, pp. 219–232. ISBN: 978-3-642-40564-8.

Čermák, Petr et al. (2014). 'MCMAS-SLK: A model checker for the verification of strategy logic specifications'. In: *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*. Vol. 8559 LNCS, pp. 525–532. ISBN: 9783319088662. DOI: `10.1007/978-3-319-08867-9_34`. arXiv: `1402.2948`.

Dastani, Mehdi et al. (2005). 'Modelling user preferences and mediating agents in electronic commerce'. In: *Knowledge-Based Systems* 18.7, pp. 335–352. ISSN: 09507051. DOI: `10.1016/j.knosys.2005.05.001`.

Edwards, Steven J. (1994). *Portable Game Notation Specification and Implementation Guide*. URL: `https://opensource.apple.com/source/Chess/Chess-110.0.6/Documentation/PGN-Standard.txt` (visited on 09/01/2018).

Gebser, Martin, Torsten Grote and Torsten Schaub (2010). 'Coala: A Compiler from Action Languages to ASP'. In: *Logics in Artificial Intelligence*. Ed. by Tomi Janhunen and Ilkka Niemelä. Berlin, Heidelberg: Springer Berlin Heidelberg, pp. 360–364. ISBN: 978-3-642-15675-5.

Gelfond, Michael and Vladimir Lifschitz (1988). *The stable model semantics for logic programming*. DOI: `10.1.1.24.6050`.

Gelfond, Michael and Vladimir Lifschitz (1998). 'Action languages'. In: *Electronic Transactions on AI* 3.16, pp. 1–23. URL: `http://ssdi.di.fct.unl.pt/rcr/geral/biblio/assets/gelfond98action.pdf`.

Grasso, G, N Leone and F Ricca (2013). *Answer set programming: Language, applications and development tools*. DOI: `10.1007/978-3-642-39666-3_3`. URL: `https://www.scopus.com/inward/record.uri?eid=2-s2.0-84881127729{\&`

}doi=10.1007{\%}2F978-3-642-39666-3{_}3{\&}partnerID=40{\&}md5=d476b39aec47180e307cc402ef5b94a1.

Gunning, David (2016). *Explainable Artificial Intelligence (XAI)*. URL: `https://www.cc.gatech.edu/{~}alanwags/DLAI2016/{\%}28Gunning{\%}29IJCAI-16DLAIWS.pdf` (visited on 28/04/2018).

Heineman, George T, Gary Pollice and Stanley Selkow (2008). 'Path Finding in AI'. In: *Algorithms in a Nutshell*, pp. 213–217. ISBN: 9780596516246. DOI: `10.1093/aje/kwq410`. URL: `http://www.ncbi.nlm.nih.gov/pubmed/21047818`.

Kaiser, Łukasz (2012). 'Learning Games from Videos Guided by Descriptive Complexity'. In: *Aaai*, pp. 963–969. URL: `http://www.aaai.org/ocs/index.php/AAAI/AAAI12/paper/viewPDFInterstitial/5091/5508`.

Kaminski, Roland (2014). *Clingo*. URL: `https://github.com/potassco/clingo`.

Law, Mark, Alessandra Russo and Krysia Broda (2014). 'Inductive Learning of Answer Set Programs'. In: *European Conference on Logics in Artificial Intelligence (JELIA)* 2.Ray 2009, pp. 311–325. ISSN: 16113349. arXiv: `1608.01946`.

Law, Mark, Alessandra Russo and Krysia Broda (2015a). 'Learning weak constraints in answer set programming'. In: *Theory and Practice of Logic Programming* 15.4-5, pp. 511–525. ISSN: 14753081. DOI: `10.1017/S1471068415000198`. arXiv: `1507.06566`.

Law, Mark, Alessandra Russo and Krysia Broda (2015b). *The ILASP system for learning Answer Set Programs*. URL: `https://www.doc.ic.ac.uk/{~}ml1909/ILASP` (visited on 22/12/2017).

Law, Mark, Alessandra Russo and Krysia Broda (2016). 'Iterative Learning of Answer Set Programs from Context Dependent Examples'. In: *Theory and Practice of Logic Programming*. Vol. 16. 5-6, pp. 834–848. DOI: `10.1017/S1471068416000351`. arXiv: `1608.01946`.

Law, Mark, Alessandra Russo and Krysia Broda (2018). *The Meta-Program Injection Feature in ILASP*. Tech. rep. Imperial College London, p. 2. URL: `https://www.doc.ic.ac.uk/{~}ml1909/ILASP/inject.pdf`.

Lee, Joohyung (2012). 'Reformulating action language C+ in answer set programming'. In: *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)* 7265, pp. 405–421. ISSN: 03029743. DOI: `10.1007/978-3-642-30743-0_28`.

Lifschitz, Vladimir (2008). 'What Is Answer Set Programming?' In: *Aaai 2008*, pp. 1594–1597.

Liu, Tie-Yan (2007). 'Learning to Rank for Information Retrieval'. In: *Foundations and Trends® in Information Retrieval* 3.3, pp. 225–331. ISSN: 1554-0669. DOI: `10.1561/1500000016`. arXiv: `arXiv:1208.5535v1`. URL: `http://www.nowpublishers.com/article/Details/INR-016`.

Love, Nathaniel et al. (2006). 'General Game Playing: Game Description Language Specification'. In: *Science* LG–2006–01. URL: `http://games.stanford.edu/language/spec/gdl{\_}spec{\_}2008{\_}03.pdf`.

Marsland, T. A. (1986). 'A review of game-tree pruning'. In: *ICCA journal* 9.1, pp. 3–19. URL: `http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.67.7446{\&}rep=rep1{\&}type=pdf`.

Mnih, Volodymyr et al. (2013). 'Playing Atari with Deep Reinforcement Learning'. In: *arXiv preprint arXiv:1312.5602*.

Muggleton, Stephen H. et al. (2014). 'Meta-interpretive learning: Application to grammatical inference'. In: *Machine Learning* 94.1, pp. 25–49. ISSN: 08856125. DOI: 10.1007/s10994-013-5358-3.

Muggleton, Stephen (1991). 'Inductive logic programming'. In: *New Generation Computing* 8.4, pp. 295–318. ISSN: 02883635. DOI: 10.1007/BF03037089.

Muggleton, Stephen (1995). 'Inverse entailment and progol'. In: *New Generation Computing* 13.3-4, pp. 245–286. ISSN: 02883635. DOI: 10.1007/BF03037227.

Muggleton, Stephen and Luc de Raedt (1994). 'Inductive Logic Programming: Theory and Methods'. In: *Journal of Logic Programming* 19.20, pp. 629–679. ISSN: 07431066. DOI: 10.1016/0743-1066(94)90035-3. URL: http://homes.soic.indiana.edu/natarasr/Courses/I590/Papers/ilp.pdf.

Muggleton, Stephen et al. (2011). 'ILP turns 20'. In: *Machine Learning* 86.1, pp. 3–23. ISSN: 0885-6125. DOI: 10.1007/s10994-011-5259-2. URL: http://www.springerlink.com/content/9463m43357074631/.

Olah, Chris et al. (2018). 'The Building Blocks of Interpretability'. In: *Distill*. DOI: 10.23915/distill.00010.

Ray, Oliver, Krysia Broda and Alessandra Russo (2003). 'Hybrid abductive inductive learning: A generalisation of Progol'. In: *Proceedings of the International Conference on Inductive Logic Programming* 2835, pp. 311–328. ISSN: 3-540-20144-0. DOI: 10.1007/978-3-540-39917-9_21. URL: http://www.springerlink.com/index/LXQFK7E5V8K3D0AV.pdf.

Romstad, Tord et al. *Stockfish*. URL: https://stockfishchess.org.

Samek, Wojciech, Thomas Wiegand and Klaus-Robert Müller (2017). 'Explainable Artificial Intelligence: Understanding, Visualizing and Interpreting Deep Learning Models'. In: arXiv: 1708.08296. URL: https://arxiv.org/pdf/1708.08296.pdfhttp://arxiv.org/abs/1708.08296.

Shortliffe, Edward H (1977). 'Mycin: A Knowledge-Based Computer Program Applied to Infectious Diseases'. In: *Proceedings of the Annual Symposium on Computer Application in Medical Care*, pp. 66–69. ISBN: 0195-4210. URL: https://www.ncbi.nlm.nih.gov/pmc/articles/PMC2464549/pdf/procascamc00015-0074.pdf{\%}0Ahttp://www.ncbi.nlm.nih.gov.proxygw.wrlc.org/pmc/articles/PMC2464549/.

Silver, David et al. (2017a). 'Mastering Chess and Shogi by Self-Play with a General Reinforcement Learning Algorithm'. In: *arXiv preprint arXiv:1712.01815*.

Silver, David et al. (2017b). 'Mastering the game of Go without human knowledge'. In: *Nature* 550.7676, pp. 354–359. ISSN: 14764687. DOI: 10.1038/nature24270. arXiv: 1610.00633.

Takizawa, Makoto. *Elmo*. URL: https://github.com/mk-takizawa/elmo{_}for{_}learn.

Tesauro, Gerald (1995). 'Temporal difference learning and TD-Gammon'. In: *Communications of the ACM* 38.3, pp. 58–68. ISSN: 00010782. DOI: 10.1145/203330.203343. URL: http://portal.acm.org/citation.cfm?doid=203330.203343.

Thielscher, Michael (2010). 'A general game description language for incomplete information games'. In: *Proceedings of the Twenty-Fourth AAAI Conference on Artificial Intelligence (AAAI-10)*, pp. 994–999. URL: https://www.aaai.org/ocs/index.php/AAAI/AAAI10/paper/view/1727.

Vinyals, Oriol et al. (2017). 'StarCraft II: A New Challenge for Reinforcement Learning'. In: *arXiv preprint arXiv:1708.04782*. DOI: `https://deepmind.com/documents/110/sc2le.pdf`. arXiv: `1708.04782`. URL: `http://arxiv.org/abs/1708.04782`.

Zhang, Dongmo and Michael Thielscher (2015). 'Representing and Reasoning about Game Strategies'. In: *Journal of Philosophical Logic* 44.2, pp. 203–236. ISSN: 15730433. DOI: `10.1007/s10992-014-9334-6`. arXiv: `1407.5380`.