The *GRL* System: Learning Board Game Rules With Piece-Move Interactions

Peter Gregory
Digital Futures Institute
School of Computing
Teesside University
Middlesbrough, UK
p.gregory@tees.ac.uk

Yngvi Björnsson and **Stephan Schiffel**
School of Computer Science
Reykjavik University
Reykjavik, Iceland
{yngvi,stephans}@ru.is

Abstract

Many real-world systems can be represented as formal state transition systems. The modelling process, in other words the process of constructing these systems, is a time-consuming and error-prone activity. In order to counter these difficulties, efforts have been made in various communities to learn the models from input data. One learning approach is to learn models from example transition sequences. Learning state transition systems from example transition sequences is helpful in many situations. For example, where no formal description of a transition system already exists, or when wishing to translate between different formalisms.

In this work, we study the problem of learning formal models of the rules of board games, using as input only example sequences of the moves made in playing those games. Our work is distinguished from previous work in this area in that we learn the interactions between the pieces in the games. We supplement a previous game rule acquisition system by allowing pieces to be added and removed from the board during play, and using a planning domain model acquisition system to encode the relationships between the pieces that interact during a move.

1 Introduction

Over the last decade, or ever since the advent of the *General Game-Playing (GGP)* competition [Genesereth *et al.*, 2005], research interest in general approaches to intelligent game playing has become increasingly mainstay. GGP systems autonomously learn how to skilfully play a wide variety of (simultaneous or alternating) turn-based games, given only a description of the game rules. Similarly, *General Video-Game (GVG)* systems learn strategies for playing various video games in real-time and non-turn-based settings.

In the above mentioned systems the domain model (i.e., rules) for the game at hand is sent to the game-playing agent at the beginning of each match, allowing legitimate play off the bat. For example, games in GGP are described in a language named *Game Description Language (GDL)* [Love *et al.*, 2006], which has axioms for describing the initial game state, the generation of legal moves and how they alter the game state, and how to detect and score terminal positions. Respectively, video games are described in the *Video Game Description Language (VGDL)* [Schaul, 2013]. The agent then gradually learns improved strategies for playing the game at a competitive level, typically by playing against itself or other agents. However, ideally one would like to build fully autonomous game-playing systems, that is, systems capable of learning not only the necessary game-playing strategies but also the underlying domain model. Such systems would learn skilful play simply by observing others play.

Automated model acquisition is an active research area spanning many domains, including constraint programming and computer security (e.g. [O'Sullivan, 2010; Bessiere *et al.*, 2014; Aarts *et al.*, 2013]). There has been some recent work in GGP in that direction using a simplified subset of board games, henceforth referred to as *Simplifed Game Rule Learner (SGRL)* [Björnsson, 2012]. In the related field of autonomous planning, the *LOCM* family of domain model acquisition systems [Cresswell *et al.*, 2009; Cresswell and Gregory, 2011; Cresswell *et al.*, 2013] learn planning domain models from collections of plans. In comparison to other systems of the same type, these systems require only a minimal amount of information in order to form hypotheses: they only require plan traces, where other systems require state information.

In this work we extend current work on learning formal models of the rules of (simplified) board games, using as input only example sequences of the moves made in playing those games. More specifically, we extend the previous *SGRL* game rule acquisition system by allowing pieces to be added and removed from the board during play, and by using the *LOCM* planning domain model acquisition system for encoding and learning the relationships between the pieces that interact during a move, allowing modelling of moves that have side effects (such as castling in chess). Our work is thus distinguished from previous work in this area in that we learn the interactions between the pieces in the games.

The paper is structured as follows: the next section provides necessary background material on *LOCM* and *SGRL*, followed by a description of the combined approach. This is followed by empirical evaluation and overview of related work, before concluding and discussing future work.

2 Background

In this section we provide background information about the LOCM and SGRL model acquisition systems that we base the present work on.

2.1 LOCM

The *LOCM* family of domain model acquisition systems [Cresswell *et al.*, 2009; Cresswell and Gregory, 2011] are inductive reasoning systems that learn planning domain models from only action traces. This is large restriction, as other similar systems require extra information (such as predicate definitions, initial and goal states, etc.). *LOCM* is able to recover domain information from such a limited amount of input due to assumptions about the structure of the output domain.

A full discussion of the *LOCM* algorithm is omitted and the interested reader is referred to the background literature [Cresswell *et al.*, 2009; Cresswell and Gregory, 2011; Gregory and Cresswell, 2015] for more information. However, we discuss those aspects of the system as relevant to this work. We use the well-known Blocksworld domain as an example to demonstrate the form of input to, and output gained from, *LOCM*. Although a simple domain, it is useful as it demonstrates a range of features from both *LOCM* and *LOCM2* that are relevant to this work.

The input to the *LOCM* system is a collection of plan traces. Suppose, in the Blocksworld domain, we had the problem of reversing a two block tower, where block A is initially placed on block B. The following plan trace is a valid plan for this problem in the Blocksworld domain:

```
(unstack A B)
(put-down A)
(pick-up B)
(stack B A)
```

Each action comprises a set of indexed object transitions. For example, the unstack action comprises a transition for block A (which we denote as *unstack.1*) and another for block B (which we denote as *unstack.2*). A key assumption in the *LOCM* algorithm is that the behaviour of each type of object can be encoded in one or more DFAs, where each transition appears at most once. This assumption means that for two object plan traces with the same prefix, the next transition for that object must exit from the same state. Consider plan trace 1 and 2 below:

```
1: (unstack A B)
1: (put-down A)

2: (unstack X Y)
2: (stack X Z)
```

In the first plan trace, block A is unstacked, before being put down on to the table. In the second plan trace, X is unstacked from block Y before being stacked on to another block, Z. The assumption that each transition only exists once within the DFA description of an object type means that the state that is achieved following an *unstack.1* transition is the same state that precedes both a *put-down.1* and a *stack.1* transition.

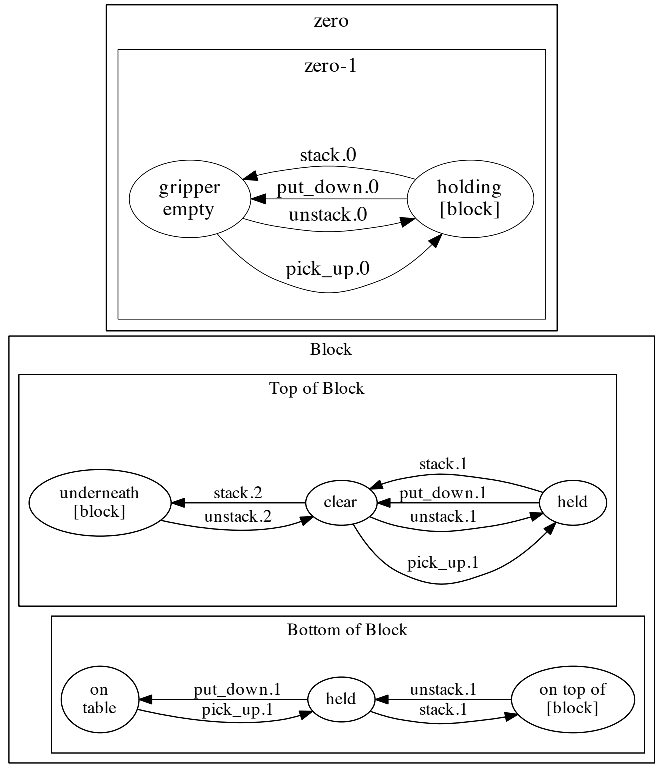

Figure 1: State machines learnt by *LOCM* in the Blocksworld planning domain. State labels are manually annotated to aid comprehension.

The output formalism of *LOCM* represents each object type as one or more parameterised DFAs. Figure 1 shows the output DFAs of *LOCM* for the Blocksworld domain. In this figure, we have manually annotated the state names in order to highlight the meanings of each state. The edges in the *LOCM* state machines represent object transitions, where each transition is labelled with an action name and a parameter position.

LOCM works in two stages: firstly to encode the structure of the DFAs, secondly to detect the state parameters.

Blocks are the only type of object in Blocksworld, and are represented by the two state machines at the bottom of Figure 1. Informally, these machines can be seen to represent what is happening above and below the block, respectively. In each planning state, a block is represented by two DFA states (for example, a block placed on the table with nothing above it would be represented by the 'clear' and 'on table' DFA states). Each of the block DFAs transition simultaneously when an action is performed, so when the previously discussed block is picked up from the table (transition pick_up.1) both the top and bottom machines move to the 'held' state.

The machine at the top of Figure 1 is a special machine known as the zero machine (this refers to an imaginary zeroth parameter in every action) and this machine can be seen as encoding the structural regularities of the input action sequences. In the case of Blocksworld, the zero machine encodes the behaviour of the gripper that picks up and puts

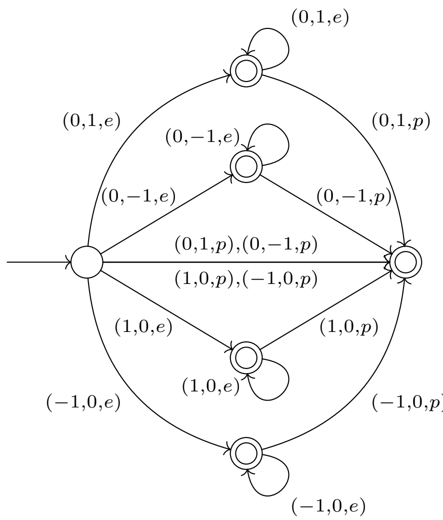

Figure 2: A DFA, D_{rook}, describing the movements of a rook in chess

down each block. Relationships between objects are represented by state parameters. As an example, in the 'Bottom of Block' machine, the state labelled 'on top of' has a block state parameter which represents the block directly underneath the current block.

2.2 The *SGRL* System

The *SGRL* approach [Björnsson, 2012] models the movements of each piece type in the game individually using a deterministic finite automata (DFA). One can think of the DFA representing a language where each word describes a completed move (or *piece movement pattern*) on the board and where each letter in the word —represented by a triplet $(\Delta x, \Delta y, on)$— describes an atomic movement. The triplet is read in the context of a game board position and a current square, with the first two coordinates telling relative board displacement from the current square (file and rank, respectively) and the third coordinate telling the content of the resulting relative square (the letter *e* indicates an empty square, *w* an own piece, and *p* an opponent's piece). For example, $(0\ 1\ e)$ indicates a piece moving one square up the board to an empty square, and the word $(0\ 1\ e)\ (0\ 1\ e)\ (0\ 1\ p)$ a movement where a piece steps three squares up (over two empty squares) and captures an opponent's piece. Figure 2 shows an example DFA describing the movements of a rook.

There are pros and cons with using the DFA formalism for representing legitimate piece movements. One nice aspect of the approach is that well-known methods can be used for the domain acquisition task, which is to infer from observed piece movement patterns a consistent DFA for each piece type (we are only concerned with piece movements here; for details about how terminal conditions and other model aspects are acquired we refer to the original paper). Another important aspect, especially for a game-playing program, is that state-space manipulation is fast. For example, when generating legal moves for a piece the DFA is traversed in a depth-first manner. On each transition the label of an edge is used to find which square to reference and its expected content. If there are no matching edges the search backtracks. A transition into a final DFA state s generates a move in the form of a piece movement pattern consisting of the edge labels that were traversed from the start state to reach s. A special provision is taken to detect and avoid cyclic square reference in piece-movement patterns.

Unfortunately, simplifying compromises were necessary in *SGRL* to allow the convenient domain-learning mechanism and fast state-space manipulation. One such is that moves are not allowed to have side effects, that is, a piece movement is not allowed to affect other piece locations or types (with the only exception that a moving piece captures the piece it lands on). These restrictions for example disallow castling, en-passant, and promotion moves in chess. We now look at how these restrictions can be relaxed.

3 The *GRL* System

In this section, we introduce a boardgame rule learning system that combines the strengths of both the *SGRL* and *LOCM* systems. One strength of the *SGRL* rule learning system is that it uses a relative coordinate system in order to generalise the input gameplay traces into concise DFAs. A strength of the *LOCM* system is that it can generalise relationships between objects that undergo simultaneous transitions.

One feature of the *SGRL* input language is that it represents the distinction between pieces belonging to the player and the opponent, but not the piece identities. This provides a trade-off between what it is possible to model in the DFAs and the efficiency of learning (far more games would have to be observed to learn if the same rules had to be learnt for every piece that can be moved over or taken, and the size of the automata learnt would be massive). In some cases, however, the identities of the interacting pieces are significant in the game.

We present the *GRL* system that enhances the *SGRL* system in such a way as to allow it to discover a restricted form of side-effects of actions. These side-effects are the rules that govern piece-taking, piece-production and composite moves. *SGRL* allows for limited piece-taking, but not the types such as in checkers where the piece taken is not on the destination square of the piece moving. Piece-production (for example, promotion in chess or adding men at the start of nine mens morris) is not at all possible in *SGRL*. Composite moves, such as castling in chess or arrow firing in Amazons also cannot be represented in the *SGRL* DFA formalism.

3.1 The *GRL* Algorithm

The *GRL* algorithm can be specified in three steps:

1. Perform an extended *SGRL* analysis on the game traces (we call this extended analysis *SGRL*+). *SGRL* is extended by adding additional vocabulary to the input language that encode the addition and removal of pieces from the board. Minor modifications have to be made to the consistency checks of *SGRL* in order to allow this additional vocabulary.
2. Transform the game traces and the *SGRL*+ output into *LOCM* input plan traces, one for each game piece.

Within this step, a single piece move includes everything that happens from the time a player picks up a piece until it is the next player's turn. For example, a compound move will have all of its steps represented in the plan trace.

3. Use the *LOCM* system to generate a planning domain model for each piece type. These domain models encode how a player's move can progress for each piece type. Crucially, unlike the *SGRL* automata, the learn domains refer to multiple piece types, and the relationships between objects through the transitions.

The output of this procedure will be a set of planning domains (one for each piece) which can generate the legal moves of a board game. By specifying a planning problem with the current board as the initial state, then enumerating the state space of the problem provides all possible moves for that piece. We now describe these three stages in more detail.

3.2 Extending *SGRL* Analysis for Piece Addition and Deletion

The first stage of *GRL* is to perform an extended version of the *SGRL* analysis. The input alphabet has been extended in order to include vocabulary to encode piece addition and removal. To this end, we add the following two letters as possible commands to the DFA alphabet:

1. The letter 'a' which means that the piece has just been added at the specified cell (the relative coordinates of this move will always be $(0, 0)$). The piece that is picked up now becomes the piece that is 'controlled' by the subsequent micro-moves.
2. The letter 'd' which means that the square has a piece underneath it which is removed from the game. This is to model taking pieces in games like peg solitaire or checkers.

The order in which these new words in the vocabulary are used is significant. An 'a' implies that the piece underneath the new piece remains in place after the move, unless it is on the final move of the sequence of micro-moves (this is consistent with the more limited piece-taking in *SGRL*, where pieces are taken only at the end of a move, if the controlled piece is another piece's square). For example, the following sequences both describe white pawn promotion in chess:

```
(0 1 e) (0 0 a) (0 0 d)
(0 1 e) (0 0 a)
```

Adding this additional vocabulary complicates the *SGRL* analysis in two ways: firstly in the definition of the game state, and secondly in the consistency checking of candidate DFAs. There is no issue when dealing with the removal of pieces and state definitions. There is, however, a small issue when dealing with piece addition. A move specified in *SGRL* input language is in the following form:

```
(12 (0 1 e) (0 0 a) (0 1 e))
```

This move specifies that the piece on square 12 moves vertically up a square, then a new piece piece appears (and becomes the controlled piece), which subsequently moves another square upwards. The issue is that for each distinct move, in *SGRL*, the controlled piece is defined by the square specified at the start of the move (in this case square 12). If a piece appears during a move, then *SGRL*+ needs to have some way of knowing which piece has appeared, in order to add this sequence to the prefix tree of the piece type. To mitigate this problem, we require all added pieces to be placed in the state before they are added, and as such they can now be readily identified. This approach allows us to learn the DFAs for all piece types, including piece addition.

The consistency algorithm of *SGRL* verifies whether a hypothesised DFA is consistent with the input data. One part of this algorithm that is not discussed in prior work is the *generateMoves* function, that generates the valid sentences (and hence the valid piece moves) of the state machine. There are two termination criteria for this function: firstly the dimensions of the board (a piece cannot move outside of the confines of the board), secondly on revisiting squares (this is important in case cycles are generated in the hypothesis generation phase). This second case is relaxed in *GRL* slightly since squares may be visited more than once due to pieces appearing and / or disappearing. However, we still wish to prevent looping behaviour, and so instead of terminating when the same square is visited, we terminate when the same square is visited in the same *state* in the DFA.

With these two changes to *SGRL* analysis we have provided a means to representing pieces that appear and that remove other pieces from the board. However, the state machines do not tell us how the individual piece movement, additions and removals combine to create a complete move in the game. For this task, we employ use of the *LOCM* system, in order to learn the piece-move interactions.

3.3 Converting to *LOCM* Input

We use the *LOCM* system to discover relationships between the pieces and the squares, and to do this we need to convert the micro-moves generated by the *SGRL*+ DFAs to sequences of actions. To convert these, we introduce the following action templates (where X is the name of one of the pieces):

```
appearX ( square )
removeX ( square )
moveX ( squareFrom, squareTo )
moveOverX ( squareFrom, squareVia, squareTo )
moveAndRemoveX ( squareFrom, squareTo )
moveAndRemoveX ( squareFrom, squareVia, squareTo )
```

These action templates mirror the input language of the extended *SGRL*+ system detailed above. Each plan that is provided as input to *LOCM* is a sequence of micro-moves that define an entire player's turn. When *LOCM* learns a model for each piece type, the interactions between the pieces and the squares they visit are encoded in the resultant domain model.

The input actions, therefore, encode the entire (possibly compound and with piece-taking side-effects) move. As an example from the Amazons[1] game, the following plan trace

[1]Amazons is a two-player game played on a 10×10 board, in which each player has four pieces. On each turn, a player moves a piece as the queen moves in chess, before firing an arrow from the destination square, again in the move pattern of the queen in chess. The final square of the arrow is then removed from the game. The loser is the first player unable to make a move.

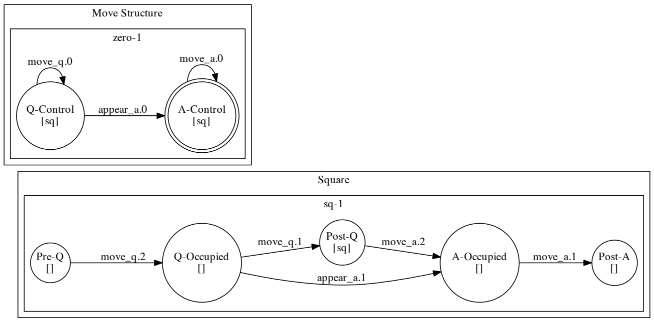

Figure 3: The *LOCM* State Machines learnt for the Queen piece in Amazons. The zero machine at the top of the figure shows the higher-level structure of the Queen's move. The Queen moves for a certain number of moves, then an arrow appears, and the arrow moves for a certain number of moves.

may be generated from the example micro-move sequence, meaning a queen is moved from A1 to B1, and then an arrow appears, and is fired to square C1:

```
Micro-move Sequence:
(0 (1 0 e) (0 0 a) (1 0 e))

Equivalent Plan Trace:
moveQueen ( A1, B1 )
appearArrow ( B1 )
moveArrow ( B1, C1 )
```

In all plan traces for the Queen piece in the Amazons game, the destination square of the Queen will always be the square in which the Arrow appears, and from which the Arrow piece begins its movement. Once the *LOCM* analysis is complete, this relationship is detected, and the induced planning domain will only allow arrows to be fired from the square that the queen moves to.

Figure 3 shows the actual *LOCM* state machines learnt for the Queen piece from a collection of game traces. The top state machine represents the zero machine, which describes the general plan structure. The machine underneath represents one of the squares on the board. The parameters of the zero machine represent the current square underneath the controlled piece. As can be seen from the generated PDDL action for $appear\_a$ (shown in Figure 4), this parameter ensures that the arrow appears at the same location that the queen ended its move on. It also ensures that the moves are sequenced in the correct way: the Queen piece moves, the Arrow piece appears and finally the Arrow piece moves to its destination. The machine at the bottom of Figure 3 encodes the transitions that a square on the board goes through during a Queen move. Each square undergoes a similar transition path, with two slightly different variants, and to understand these transitions it is important to see the two options of what happens on a square once a queen arrives there. The first option is that the Queen transits the square, moving on to the next square. The second option is that the Queen ends its movement; in this case, the Arrow appears, before transiting the square. These two cases are taken into account in the two different paths through the state machine.

As another example, consider the game of Checkers. The state machines generated by *LOCM* for the Pawn piece type

```
(:action appear_a
 :parameters (?Sq1 - sq)
 :precondition
   (and (Q_Control ?Sq1)
        (Q_Occupied ?Sq1))
   :effect
   (and (A_Control ?Sq1)
        (not (Q_Control ?Sq1))
        (A_Occupied ?Sq1)
        (not (Q_Occupied ?Sq1))))
```

Figure 4: The PDDL Action for `appear_a`, representing the arrow appearing in the Amazons game.

are shown in Figure 5. There are two zero machines in this example, which in this case means that *LOCM2* analysis was required to model the transition sequence, and there are separable behaviours in the move structure. These separable behaviours are the movement of the Pawn itself and of the King, if and when the Pawn is promoted. The top machine models the alternation between taking a piece and moving on to the next square for both Pawns and Kings, the bottom machine ensures that firstly a King cannot move until it has appeared, and secondly that a Pawn cannot continue to move once it has been promoted.

3.4 Move Generation

We now have enough information to generate possible moves based on a current state. The *GRL* automata can be used in order to generate moves, restricted by the *LOCM* state machines, where the *LOCM* machines define the order in which the pieces can move within a single turn.

The algorithm for generating moves is presented here as Algorithm 1. The algorithm takes as input an *SGRL*+ state, a *LOCM* state and a square on the board. A depth-first search is then performed over the search space, in order to generate all valid traces. A state in this context is a triple of a board square, an *SGRL*+ piece state and a *LOCM* state. Lines 2 to 3 define the termination criteria (a state has already been visited), lines 4 to 5 define when a valid move has been defined (when each of the *SGRL*+ and *LOCM* DFAs are in a terminal state), and lines 6 to 7 define the recursion (in these lines, we refer to a state as being consistent. Consistency in this sense means that the state is generated by a valid synchronous transition in both the *SGRL*+ and *LOCM* DFAs.

Algorithm 1 Function to generate legal moves for a player from a combination of *SGRL*+ and *LOCM* DFAs.

1: **function** GenerateMoves(sq,$\mathrm{S_L}$,$\mathrm{S_P}$)
2:     **if** Visited $\langle sq, \mathrm{S_L}, \mathrm{S_P}\rangle$ **then**
3:         **return**
4:     **if** terminal($\mathrm{S_L}$) and terminal($\mathrm{S_P}$) **then**
5:         add the current micro-move path to moves.
6:     **for all** consistent next states $\langle sq', S'_L, S'_P\rangle$ **do**
7:         GenerateMoves(sq', $\mathrm{S'_L}$,$\mathrm{S'_P}$)

This completes the definition of the *GRL* system: an extended *SGRL* learns DFAs for each piece type, the *LOCM* system learns how these piece types combine to create entire moves, and then the move generation algorithm produces the

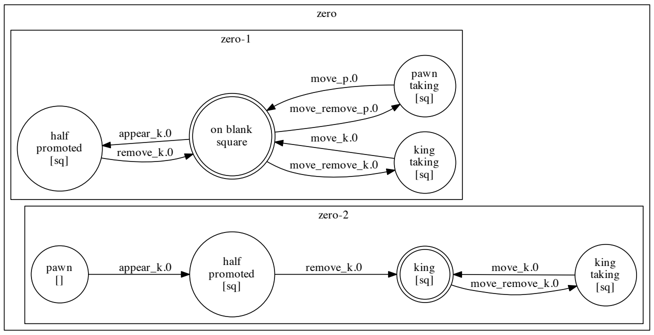

Figure 5: The *LOCM* State Machines learnt for the Pawn piece in the game of Checkers. The state machine incorporates the possibility of the Pawn being promoted into a King, and then subsequently can take other pieces.

valid moves in any given state. Next, we provide an evaluation of *GRL*, and detail the cost of each element of the system.

4 Empirical Evaluation

In this section, we provide an evaluation of the *GRL* system. In order to perform this evaluation, we learn the rules of Amazons, Peg Solitaire and Checkers. We also provide evaluation for the three games Breakthrough, Breakthrough Checkers, and Breakthrough Chess as used in the *SGRL* evaluation [Björnsson, 2012] in order to demonstrate that performance is not adversely affected due to the changes in *GRL*.We report on the time taken in the extended *SGRL+* phase, and the *LOCM* phase, to show the balance of time taken in each phase. We generate two forms of game traces for each game: one has a single move per turn (for these we generate 1000 game traces) and the other enumerates all possible moves (for these we generate 50 game traces). This method of evaluation is consistent with that chosen in the prior evaluation of *GRL*. All experiments are run on Mac OSX 10.10, running on an Intel i7 4650U CPU, with 8GB system RAM.

Table 1 shows the results of *GRL* on our benchmark problems. We show the time taken for each element of *GRL* to learn its model. We report both the time taken to learn the model for the first and second players (reported under ‘Player1’ and ‘Player2’). In the case of Peg Solitaire, there is only one player, which explains the missing data. The missing data in the *LOCM* element for Amazons A piece is because an arrow piece in Amazons never starts a move itself and hence has no plan trace data. We first observe that the Breakthrough, Breakthrough Checkers and Breakthrough Chess results are not significantly different to the results for *SGRL*. Therefore the modifications made to the *SGRL* algorithm are not adversely affecting performance. We do not report *LOCM* time here, since these games do not require the *LOCM* analysis, as *SGRL+* is sufficient to represent them. It is notable that the time taken to learn the *LOCM* models is significantly larger than the time to learn the *SGRL+* individual piece models. The main cause of this difference is that the input plan traces are for every *turn* of the game, rather than the entire game trace. Because of this, *LOCM* has 25,000 input traces for the Queen piece in Amazons and 11,000 for

Game	Piece	Element	Player1	Player2
Amazons	Q	*SGRL+*	8.1	35.2
		LOCM	20.9	30.0
	A	*SGRL+*	9.3	10.8
		LOCM	–	–
Checkers	P	*SGRL+*	<0.01	<0.01
		LOCM	260.4	275.3
	K	*SGRL+*	<0.01	<0.01
		LOCM	52.9	47.0
Peg Solitaire	P	*SGRL+*	<0.01	–
		LOCM	8.8	–
BT	P	*SGRL+*	<0.01	<0.01
CheckerBT	P	*SGRL+*	0.16	0.17
ChessBT	P	*SGRL+*	<0.01	<0.01
	K	*SGRL+*	<0.01	<0.01
	N	*SGRL+*	<0.01	<0.01
	B	*SGRL+*	3.3	3.4
	R	*SGRL+*	3.9	3.8
	Q	*SGRL+*	12.3	12.5

Table 1: Learning time in seconds to learn models for each of the pieces in the problem set with all moves known. (Using all moves only affects the time for the *SGRL+* element, as *LOCM* does not accept input when all moves are known).

the Pawn piece in Checkers, for example. The time required to parse and analyse this number of plans is necessarily time consuming.

Table 2 shows the results of learning when only a single move per turn is known. Learning times are increased, as it takes longer to find consistent DFAs in the *SGRL+* phase. Note that on several occasions, *SGRL+* returns incorrect solutions. In these cases, there is simply insufficient input data. *GRL* is still effective in the majority of cases, however, and does not degrade the performance of *SGRL* on the previous game data.

5 Related Work

Within the planning literature there are several domain model acquisition systems, each with varying levels of detail in their input observations. The *Opmaker2* system [McCluskey *et al.*, 2009; Richardson, 2008] learns models in the target language of OCL [McCluskey and Porteous, 1997] and requires a partial domain model, along with example plans as input. The ARMS system [Wu *et al.*, 2007], can learn STRIPS domain models with partial or no observation of intermediate states in the plans, but does at least require predicates to be declared. The LAMP system [Zhuo *et al.*, 2010] can target PDDL representations with quantifiers and logical implications.

As for domain model acquisition in boardgames, an ILP approach exists for inducing chess variant rules from a set of positive and negative examples using background knowledge and theory revision [Muggleton *et al.*, 2009]. Furthermore, [Kaiser, 2012] presents a system that learns games such as Connect4 and Breakthrough from video demonstrations us-

Game	Piece	Element	Player1	Player2
Amazons	Q	*SGRL+*	51.2*	54.9
		LOCM	20.9	30.0
	A	*SGRL+*	86.2*	34.6*
		LOCM	–	–
Checkers	P	*SGRL+*	<0.01	<0.01
		LOCM	260.4	275.3
	K	*SGRL+*	<0.01	<0.01
		LOCM	52.9	47.0
Peg Solitaire	P	*SGRL+*	<0.01	<0.01
		LOCM	8.8	–
BT	P	*SGRL+*	<0.01	<0.01
CheckerBT	P	*SGRL+*	1.2*	3.1*
ChessBT	P	*SGRL+*	<0.01	<0.01
	K	*SGRL+*	<0.01	<0.01
	N	*SGRL+*	<0.01	<0.01
	B	*SGRL+*	43.2	41.0
	R	*SGRL+*	52.4	50.0
	Q	*SGRL+*	145.3	140.4

Table 2: Learning time in seconds to learn models for each of the pieces in the problem set with only a single move known per state. Results marked with a (*) returned an incorrect DFA.

ing minimal background knowledge. Rosie [Kirk and Laird, 2013] is an agent implemented in Soar that learns game rules (and other concepts) for simple board games via restricted interactions with a human. As for general video-game playing, a neuro-evolution algorithm showed good promise playing a large set of Atari 2600 games using little background knowledge [Hausknecht *et al.*, 2014].

6 Conclusions and Future Work

In this work, we presented a system capable of inducing the rules of more complex games than the current state-of-the-art system. This can help in both constructing the rules of games, or replacing the current move generation routine of an existing general game playing system (for fitting games). *GRL* improves *SGRL* by allowing piece addition, removal and structured compound moves. This was achieved by combining two techniques for domain-model acquisition, one rooted in game playing and the other in autonomous planning.

However, there remain interesting classes of board game rule that cannot be learnt by *GRL*. One interesting rule class is that of a state-bound movement restriction. Games that exhibit this behaviour allow only a subset of moves to occur in certain contexts: examples of this are check in chess (where only the subset of moves that exit check are allowed) and the compulsion to take pieces in certain varieties of checkers (thus restricting the possible moves to the subset that take pieces when forced). An approach to learning these restrictions could be developed given knowledge of all possible moves at each game state in the game.

Learning the termination criteria of a game is also an important step, if the complete set of rules of a board game are to be learnt. This requires learning properties of individual states, rather than state transition systems. However, many games have termination criteria of the type that only one player (or no players) can make a move. For this class of game, and others based on which moves are possible, it should be possible to extend the *GRL* system to learn how to detect terminal states.

References

[Aarts *et al.*, 2013] Fides Aarts, Joeri De Ruiter, and Erik Poll. Formal Models of Bank Cards for Free. In *2013 IEEE Sixth International Conference on Software Testing, Verification and Validation Workshops*, pages 461–468. Ieee, March 2013.

[Bessiere *et al.*, 2014] Christian Bessiere, Remi Coletta, Abderrazak Daoudi, Nadjib Lazaar, Younes Mechqrane, and El Houssine Bouyakhf. Boosting Constraint Acquisition via Generalization Queries. In *ECAI*, 2014.

[Björnsson, 2012] Yngvi Björnsson. Learning Rules of Simplified Boardgames by Observing. In *ECAI*, pages 175–180, 2012.

[Cresswell and Gregory, 2011] Stephen Cresswell and Peter Gregory. Generalised Domain Model Acquisition from Action Traces. In *International Conference on Automated Planning and Scheduling*, pages 42 – 49, 2011.

[Cresswell *et al.*, 2009] Stephen Cresswell, Thomas Leo McCluskey, and Margaret Mary West. Acquisition of object-centred domain models from planning examples. In Alfonso Gerevini, Adele E. Howe, Amedeo Cesta, and Ioannis Refanidis, editors, *ICAPS*. AAAI, 2009.

[Cresswell *et al.*, 2013] SN Cresswell, TL McCluskey, and MM West. Acquiring planning domain models using LOCM. *The Knowledge Engineering Review*, 28(2):195 – 213, 2013.

[Genesereth *et al.*, 2005] Michael R. Genesereth, Nathaniel Love, and Barney Pell. General Game Playing: Overview of the AAAI competition. *AI Magazine*, 26(2):62–72, 2005.

[Gregory and Cresswell, 2015] Peter Gregory and Stephen Cresswell. Domain Model Acquisition in the Presence of Static Relations in the LOP System. In *International Conference on Automated Planning and Scheduling*, pages 97–105, 2015.

[Hausknecht *et al.*, 2014] Matthew J. Hausknecht, Joel Lehman, Risto Miikkulainen, and Peter Stone. A neuroevolution approach to general atari game playing. *IEEE Trans. Comput. Intellig. and AI in Games*, 6(4):355–366, 2014.

[Kaiser, 2012] Lukasz Kaiser. Learning games from videos guided by descriptive complexity. In Jörg Hoffmann 0001 and Bart Selman, editors, *AAAI*, pages 963–970. AAAI Press, 2012.

[Kirk and Laird, 2013] James R. Kirk and John Laird. Interactive task learning for simple games. In *Advances in Cognitive Systems*, pages 11–28. AAAI Press, 2013.

[Love *et al.*, 2006] Nathaniel Love, Timothy Hinrichs, and Michael Genesereth. General Game Playing: Game description language specification. Technical Report April 4 2006, Stanford University, 2006.

[McCluskey and Porteous, 1997] Thomas Lee McCluskey and Julie Porteous. Engineering and compiling planning domain models to promote validity and efficiency. *Artificial Intelligence*, 95(1):1–65, 1997.

[McCluskey *et al.*, 2009] T. L. McCluskey, S. N. Cresswell, N. E. Richardson, and M. M. West. Automated acquisition of action knowledge. In *International Conference on Agents and Artificial Intelligence (ICAART)*, pages 93–100, January 2009.

[Muggleton *et al.*, 2009] Stephen Muggleton, Aline Paes, Vítor Santos Costa, and Gerson Zaverucha. Chess revision: Acquiring the rules of chess variants through FOL theory revision from examples. In Luc De Raedt, editor, *ILP*, volume 5989 of *LNCS*, pages 123–130. Springer, 2009.

[O'Sullivan, 2010] B O'Sullivan. Automated Modelling and Solving in Constraint Programming. *AAAI*, pages 1493–1497, 2010.

[Richardson, 2008] N. E. Richardson. *An Operator Induction Tool Supporting Knowledge Engineering in Planning*. PhD thesis, School of Computing and Engineering, University of Huddersfield, UK, 2008.

[Schaul, 2013] Tom Schaul. A video game description language for model-based or interactive learning. In *CIG*, pages 1–8. IEEE, 2013.

[Wu *et al.*, 2007] Kangheng Wu, Qiang Yang, and Yunfei Jiang. ARMS: An automatic knowledge engineering tool for learning action models for AI planning. *Knowl. Eng. Rev.*, 22(2):135–152, 2007.

[Zhuo *et al.*, 2010] Hankz Hankui Zhuo, Qiang Yang, Derek Hao Hu, and Lei Li. Learning complex action models with quantifiers and logical implications. *Artif. Intell.*, 174:1540–1569, December 2010.