Towards Inductive Logic Programming for Game Analysis: Leda

Adam Summerville
University of California, Santa Cruz
adamsumm@gmail.com

Abstract

Game generation and analysis has commonly relied on hand authored rules and heuristics. This authoring task comes with a high authorial burden, both in the amount of rules and heuristics that need to be authored for decent coverage and in the complexity of authoring these rules. In this paper I present early work on Leda and inductive logic programming system designed to learn these rules, so as to support further generation and analysis. I present Leda, describe its process, and finally show a sample set of the rules that it learns.

1 Introduction

Automatic game generation targeting specific meaning is a challenging problem, that has often run into a knowledge engineering problem. Approaches such as Game-O-Matic (Treasor et al. 2012) or the proceduralist reading system of (Martens et al. 2016) have relied on authored micro-rhetorics (meaning → game code snippets) or proceduralist inference rules (game code → meaning), respectively, as the underpinnings of their systems. These systems are limited by the types of meaning they can express/understand, and it remains an authoring challenge to expand their knowledge base.

It is with this in mind that we present Leda, an Inductive Logic Programming (ILP) system designed to take in game descriptions (sets of definitions about entities, resource, and mechanics) and observed higher order inferences in those games (“The player controls X”, “X chases Y”, etc.) and learns rules to make those inferences. The goal is to learn the kind of inference rules found in the proceduralist reading system of Martens et al. (Martens et al. 2016) with an eventual goal of reducing the burden of authoring these rules. Currently, Leda can learn relatively low level inference rules, but in the future I hope to support the idea of predicate invention, the process of creating sub-rules, so as to automatically learn higher order inference rules. Leda is a novel ILP system that produces optimal rules for its given domain, AnsProlog rules with arbitrary functors and clause negation, and I present some preliminary results of the rules that it can learn.

2 Related Work

The work of Martens et al. (Martens et al. 2016) is the biggest inspiration for this work. They used a domain specific language, Cygnus, and authored rules that could take in a game definition and produce higher order inferences about those games, ranging from simple inferences about what entities the player controls, player modeling aspects such as predicting which outcomes the player will try to achieve, up to higher order inferences such as the fact that Kaboom! is a hopeless game because lives decrease monotonically and difficulty (as expressed by the speed of the bomber) increases monotonically. It is with these sorts of inferences in mind that we showcase Leda and its rule learning capabilities. Guzdial et al. (Guzdial, Li, and Riedl 2017) learn game mechanic rules from observation. The rules are state transitions (e.g., “entity X moves left by 5 pixels”) that are learned greedily. This learning process only learns these state transitions and is unable to determine other facts about the game (e.g., player control is incorporated only as a stochastic factor that might alter the progression of rules, not as a fact about how the player controls the game). Summerville et al. (Summerville et al. 2017) learned “causal affordance” clusterings from observation, learning to group entities based on how they interacted with other entities, learning groupings that humans would label as “solid”, “harmful”, etc., but as with the work of Guzdial these groupings were purely mechanical, devoid of human level inferences. All of these fall under what Osborn et al. term Automated Game Design Learning (AGDL) (Osborn, Summerville, and Mateas 2017). I hope that the work of Leda points to another direction for AGDL that is not directly focused on learning mechanical facts about games, but rather higher order inferences that a human might make about a game.

ILP has been an active area of research over the last three decades, looking at learning logic programs from a base set of facts. Early works were focused on learning Horn clauses with no literals (e.g. Not color(red) but instead color(X), red(X)) and no functors (e.g. No color(red(X))). Early work such as Quinlan’s FOIL (Quinlan 1990) used greedy search to learn such clauses, but the greedy nature made no guarantees about whether an optimal solution would be found. Progol (Muggleton 1995) used A∗ with a heuristic of “compression”, minimizing the number of predicates used in the resulting rules. Unlike FOIL,
Prolog limits its search space via a type system, where each term has an associated type and rules are bound over type structures. GOLEM (Muggleton, Feng, and others) operates by randomly selecting pairs of positive examples and calculates the relative least general generalization of those pairs. It then keeps the clause that covers the largest number of positive examples and is consistent with the negative examples. It then keeps that clause and repeats this process until all positive examples are covered. Unlike FOIL and Progol, GOLEM is able to learn clauses with functors, but the random, greedy search precludes guarantees of global optima.

More recent work has used predicate invention whereby learned clauses can invent new predicates not found in the background knowledge. MetagolAI (Cropper and Muggleton 2016) can invent predicates via a set of meta-rules that govern how these predicates might be formed (e.g., chaining $P(X,Z) \Rightarrow Q(X,Y), R(Y,Z)$). However, it is up to the end-user to determine which meta-rules should be used, and it remains an open problem as to what meta-rules should be used (or possibly authored if the desired rule does not exist).

3 Approach

In creating Leda, I wanted an approach that satisfied:

1. Could learn arbitrary clauses (both functors and literals)
2. Required only the base background knowledge and positive examples
3. Could utilize language constructs from AnsProlog

Other approaches fail these requirements for a variety of reasons: FOIL is inappropriate due to (1), GOLEM is inappropriate due to (3), Metagol is inappropriate due to (2) and (3), etc. This led to the creation of Leda. I note that Leda is very much a work in progress, so while fully utilizing the language constructs specific to AnsProlog is a desired end goal, currently Leda only utilizes the constructs of AnsProlog shared with Prolog (e.g., aggregations such as $X \{p(Z)\} \ Y$ remain unused).

Leda takes in a list of games $G$ and a corresponding list of positive examples for those games $E$. Games take the form of Cygnus (Martens et al. 2016) programs. An example rule from a Cygnus representation of Pong is show below:

\[
\text{Rule:} \leftarrow \text{precondition}((\text{control\_event} (\text{player\_input}(\text{up\_arrow, held})), \text{move\_up}), \text{results}(\text{move\_up}, \text{moves}(\text{paddle\_player, north, low})).
\]

This rule defines the set of preconditions under which it fires (e.g., the control event coming from the player input of the Up arrowing being held down) and the results (e.g., the entity labeled as the paddle_player moves north by a low amount). All of the definitions and rules describing a game are typically on the order of 40-50 lines of Cygnus code (e.g., Pong can be represented in 37 lines and Kaboom! in 40).

Leda operates by reading in each game and determining a graph structure for how the lines of code relate to each other. This is done by determining the terms found in each line and associating lines that have the same terms. For instance, in this example the first line has the terms:

\{up\_arrow, held, move\_up\}

and the second:

\{move\_up, paddle\_player, north, low\}

So the association between these lines is that both contain the term paddle_player. This is important, as it tells us how we can expect to use these lines in our rules. For instance, we know that a rule might take the form of:

\[
\text{Rule:} \leftarrow \text{precondition}((\text{control\_event} (\text{player\_input}(A,B)), C), \text{result}(C, \text{moves}(D,E,F)).
\]

i.e. we use the only related term to determine the implicit type structure as opposed to explicitly defining the types of terms.

Given the graphical structure of how the clauses relate to each other, we now perform a breadth-first search over the rule space. The starting point of this search are the positive examples given. We will keep with the Pong motivating example and say that our positive example is player_controls(paddle_player). Given this, we know that the neighbors of player_controls(paddle_player) should include the term paddle_player. In this case, the clauses with paddle_player are:

\[
\text{entity}(\text{paddle\_player}).
\]

\[
\text{initialize}(\text{paddle\_player}, 1, \text{location}(\text{middle, left})).
\]

\[
\text{initialize}(\text{set\_sprite}(\text{paddle\_player, rectangle})).
\]

\[
\text{initialize}(\text{set\_color}(\text{paddle\_player, white})).
\]

\[
\text{moves}(\text{paddle\_player, north, low}).
\]

\[
\text{moves}(\text{paddle\_player, south, low}).
\]

\[
\text{precondition}(\text{overlaps}(\text{ball, paddle\_player, player\_hit})).
\]

Given the bias that better rules are those that are more compact, we visit the neighbors in ascending order number of predicates. With a set of clauses to be used in the rule, we then enumerate all possible rules given those clauses. This combinatorial space is made of all combinations of:

- Concrete and variable terms admitted by the clauses
- Subsumation of functors into variables
- Safe negations

Again, given the two clauses:

\[
\text{Rule:} \leftarrow \text{precondition}((\text{control\_event} (\text{player\_input}(\text{up\_arrow, held})), \text{move\_up}), \text{result}(\text{move\_up}, \text{moves}(\text{paddle\_player, north, low})).
\]

The abstractification process for the rule finds the set of all concrete terms found in the clauses (e.g., \{up\_arrow, held, move\_up, paddle\_player, north, low\}) and finds the possible combinations of variables and concrete atoms ($2^k$ in this case). As with the bias that fewer terms are more preferred, I assume that better rules will use a smaller number of concrete term. For instance, in the case of Pong, the rule:

\[
\text{player\_controls}(\text{paddle\_player}) \leftarrow \text{entity}(\text{paddle\_player}).
\]
would have full coverage of the positive examples in *Pong* but would do a poor job of covering any other game (except for say, *Breakout*, maybe). As such it is considered after the rule:

```
player_controls(X) :-
    entity(X).
```

Given all possible combinations of concrete and variable terms, we then consider whether a functor can safely subsume its terms and be considered. Continuing with the motivating example of:

```
precondition(control_event(
    player_input(up_arrow, held)), move_up).
result(move_up,
    moves(paddle_player, north, low)).
```

we see the functors:

```
control_event(player_input(up_arrow, held))
player_input(up_arrow, held)
moves(paddle_player, north, low)
```

A functor can safely be subsumed if it does not orphan any of the clauses considered. In this case, each could safely be subsumed, as the only term that ties the clauses together is `move_up`. For instance, the subsumption of

```
control_event(player_input(up_arrow, held))
player_input(up_arrow, held)
moves(paddle_player, north, low)
```

leads to the rule:

```
precondition(X), move_up.
result(move_up,
    moves(paddle_player, north, low)).
```

However, if we were to consider the clauses:

```
result(move_up,
    moves(paddle_player, north, low))
precondition(overlaps(ball, paddle_player),
    we see that neither
moves(paddle_player, north, low)
or overlaps(ball, paddle_player) could be subsumed in this case, as they both contain the linking term `paddle_player`.

Finally, we wish to be able to create rules with more expressivity than just `P ← A ∧ B ∧ ...`, and wish to consider rules that allow for the negation of clauses, i.e. `P ← A ∧ B ∧ ¬C ∧ ¬D ∧ ...`. However, we need to ensure that negated clauses remain safe constructs, i.e. they do not contain any variables that are found in any other clause. For instance the rule:

```
P(X) :-
    Q(X),
    not R(Y).
```

is unsafe as `Y` is not contained elsewhere, but

```
P(X) :-
    Q(X, Y),
    not R(Y).
```

is safe because `Y` is found in `Q(X, Y)`. As such, we only allow the negation of a clause if all variables in the clause are contained in at least one positive clause.

After finding all possible combinations of rules involving the abstractification of concrete terms, functor subsumption, and clause negation, we need to test the rules. At each point in the search we test the rule on each pair of game and positive examples found in that game. If a rule produces false positives it is ignored, and the rule of minimal complexity that covers the most positive examples per game is kept. This minimal complexity is determined by a lexicographic sorting, first sorting by number of concrete terms contained and then sorting by the number of terms found in the rule. If at any point a rule covers all positive examples in a game, then that rule is kept and the game is discarded. The search continues until all games are removed or a maximal depth is reached.

## 4 Preliminary Results

Work on *Leda* is still in an early stage; however, *Leda* is able to produce low level inference rules important for higher level inferences. I tested *Leda* on two games written in Cygnus, *Pong* and *Kaboom!*, over four low level learning tasks, determining why a given entity would be labeled as being controlled by the player, why an entity would be labeled as static, why a given outcome would be labeled as being controlled by the player, and why an entity would be labeled as being controlled by the computer. The rules that *Leda* learned are:

```
player_controls(X) :-
    precondition(control_event(C), O),
result(O, moves(X, S, D)).
```

i.e. “The player controls X if there is a precondition predicate on a control event that results in that entity moving in some way.”

```
on_static(X) :-
result(O, moves(X, S, D)).
```

i.e. “An entity is non-static if it moves in some way.”

While *Leda* can not currently perform predicate invention, by utilizing previously learned rules it can learn rules such as:

```
player_controls_outcome(O) :-
    precondition(control_event(C), O).
player_controls_outcome(O) :-
    precondition(overlaps(X, Y), O),
player_controls(X).
```

i.e. “The player controls outcome O if either that outcome is predicated on a control event or collision involving an entity X that the player controls.”

```
computer_controls(X) :-
result(O, moves(X, S, D),
not player_controls(X).
```

“The computer controls entity X if it moves in some way and the player does not control X.”

While these rules are all quite simple, they are the building blocks for higher order inferences, and it points towards promising future directions for *Leda*.

## 5 Conclusion and Future Work

I have presented early work on *Leda* an Inductive Logic Programming tool aimed at learning inference rules to extract game knowledge from games. *Leda* operates by first determining the possible structures of rules given games specified in Cygnus, iterating over them in order of increasing complexity, and testing them for how well they cover the given positive examples. Learned rules can have arbitrary functors and negation and the learning process requires no input as to
the form the learned rules must take. I have presented a small set of the rules learned by Leda, demonstrating a range of its capabilities (functor subsumption and clause negation).

In future work, I would like to be able to utilize additional AnsProlog language constructs such as counting (e.g., A = \{p (X, Y, Z)\}, A < 4.) and variable relationships (e.g., moves (X, S, D), moves (Y, S, D), X != Y.). Also, while the exhaustive search is not inordinately time consuming (all of the presented rules were learned in fewer than 2 minutes each), I would like to reduce the search space or include heuristics to reduce the search time. I would also like to allow for predicate invention so that rules such as computer_controls(X) and player_controls_outcome(O) could be learned without the need for the intermediate player_controls(X) to be learned first.

References


Guzdial, M.; Li, B.; and Riedl, M. O. 2017. Game engine learning from video. In International Joint Conference on Artificial Intelligence.


Treanor, M.; Blackford, B.; Mateas, M.; and Bogost, I. 2012. Game-o-matic: Generating videogames that represent ideas. In Second PCG workshop.