Investigating the Utility of Answer Set Programming and Inductive Logic Techniques in Learning Two-Player Game Strategies

Susana Hahn, Atreya Shankar
{hahnmartinlu,shankar}@uni-potsdam.de
PM: Computational Intelligence, WiSe 2019/20
Prof. Dr. Torsten Schaub
Research Group: Knowledge Processing and Information Systems
Institute of Computer Science, University of Potsdam

April 3, 2020

Abstract

In the last decades, significant advancements have been made in optimizing artificially intelligent computer agents in two-player games. In this project, we aim to learn strategies from games encoded in Game Description Language (GDL) while simulating the gameplay using Answer Set Programming (ASP) in *clingo*. We reformulate two common games, specifically *Nim*, and *Tic-Tac-Toe* and implement various techniques for learning explainable strategies. These approaches include flavours of the *Minimax* algorithm to generate not only optimum decision trees but also ASP rules. Further, we use Inductive Learning Programming and the framework *ILASP* to learn weak constraints from ordered examples. Finally, we benchmark the various learning techniques against one another in order to evaluate their effectiveness and scalability into bigger game instances. We were able to find strategies in the form of logic rules easily interpretable by humans. Our method represents a novel technique since we attempt to extend the utility of ASP from its traditional single-agent problem-solving setting to that of a dynamic dual-agent simulation setting.

Contents

1 Introduction

In the last decades, significant advancements have been made in optimizing artificially intelligent computer agents in two-player games such as *Chess* and *Go* (Silver et al., 2018). Much of this success can be attributed to advancements in deep reinforcement learning and corresponding GPU hardware-acceleration (Silver et al., 2018). However, one common disadvantage of these techniques is the lack of explainability of the strategies learned; which is largely due to the black-box nature of deep learning models (Verma et al., 2018). These models are specially designed for games with a high branching factor which makes it impossible to explore the complete game space to make a decision. Most of these systems rely on Monte Carlo Tree Search and Reinforcement learning techniques to avoid the construction of a full search tree. In the case of smaller games, where we are able to compute the full tree search for smaller instances, more algorithms become available to perform the analysis.

In this project, we aim to learn game strategies for two common small games, specifically *Nim*, and *Tic-Tac-Toe*. We use formalisms from Game Description Language (GDL) (Love et al., 2008) to encode the rules of the games in Answer Set Programming (ASP)(Anger et al., 2005), and simulate the gameplay using the ASP system *clingo*. We implement various techniques for learning game strategies; a brute force *Minimax* algorithm, a specialized Minimax algorithm employing the optimization statements provided by *clingo* and we also use Inductive Learning of Answer Set Programs (*ILASP*) to learn weak constraints from positive ordered examples. Finally, we benchmark the various learning techniques against one another by treating them as independent agents to evaluate their performance and scalability. Our method represents a novel technique since we attempt to extend the utility of ASP from its traditional single-agent problem-solving setting to that of a dynamic dual-agent simulation setting.

In sections 2 and 3, we describe the aforementioned background concepts and methodologies respectively. In section 4, we describe the results of our methodologies and then in section 5 proceed to discuss pertinent limitations of the tested methodologies. Finally, we conclude our research in section 6.

2 Background Concepts

2.1 Our Two-Player Games

2.1.1 Nim

Nim is an old and well-known two-player game, which is rumored to have originated in China (Bouton, 1901). Nim consists of a configuration of N piles where each pile i consists of N_i objects; which are typically referred to as counters. During each round, a single player must take at least one counter from a single pile. The players play alternately and in the normal version of the game, the player who takes the last counter(s) wins. The game is usually played with four piles with initial counters $(7, 5, 3, 1)$. Since Nim has no draws, there is always a clear winning strategy for one of the players. Such a strategy is based on performing a binary digital sum, called *Nim Sum*, representing the counters of each heap in binary and

adding each unit individually. The key is to make sure you always leave the next player in a configuration where the *Nim Sums* for every digit are even. In Figure 1 we can notice how player B uses this strategy and leaves piles with an even *Nim Sums* for every digit, unlike the moves performed by player A.

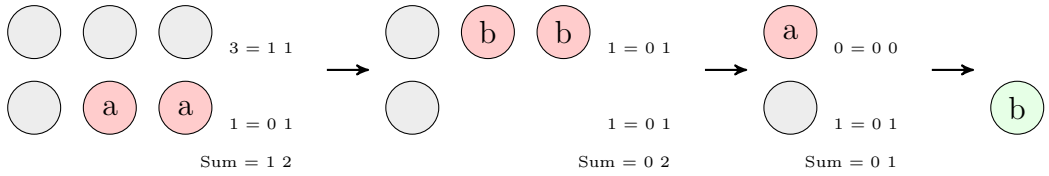

Fig. 1. Sample game of *Nim* with a 2-pile starting configuration where player B wins by taking last counter; red indicates counter(s) taken. The binary representation is always done wrt the piles left.

2.1.2 Tic-Tac-Toe

Tic-Tac-Toe is a two-player grid-based game that can be generalized to a (m, n, k) type game (Abu Dalffa et al., 2019). Two players play on a $m \times n$ size grid alternately and place an object, such as a circle and cross which is unique to each player, on the grid during each turn. A player wins the game by placing k of its own objects consecutively either in a row, column or a diagonal. If all cells in the grid are occupied without any player winning, then the game is considered to be a draw. Tic-Tac-Toe is commonly played as a $(m, n, k) \equiv (3, 3, 3)$ type game, and we will use this version in our paper. This game has no winning strategy, when both players follow the right strategy it will always finish in a draw.

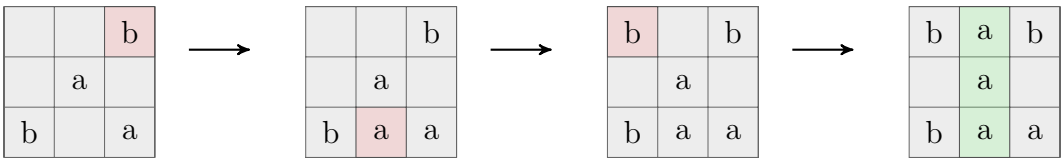

Fig. 2. Sample game of (3,3,3) *Tic-Tac-Toe* where player A wins by placing 3 consecutive objects a column; green indicates the winning column and red indicates the move made.

2.2 Game Description Language (GDL)

Game Description Language (GDL), is a declarative logic programming language (Genesereth et al., 2005). It has been developed as a high-level knowledge representation formalism for axiomatising the rules of any game. Since our project attempts to model different games using a common framework, we use the formalizations of GDL to provide a uniform means of describing these games. There are ten game-independent relation constants defined in GDL, the ones with a special interest for us are the following.

- *role(a)* means that a is a role in the game.
- *true(p)* means that the proposition p is true in the current state.
- *does(r,a)* means that player r performs action a in the current state.
- *next(p)* means that the proposition p is true in the next state.

- *legal(r,a)* means it is legal for role r to play action a in the current state.
- *goal(r,n)* means that player the current state has utility n for player r.
- *terminal* means that the current state is a terminal state.

2.3 Answer Set Programming (ASP)

According to Lifschitz (2019):

> "Answer set programming (ASP) is a form of declarative programming oriented towards difficult, primarily NP-hard, search problems."

Answer Set Programming emerged in the early 1990s as a new declarative programming paradigm (Gelfond and Lifschitz, 1991). It is based in nonmonotonic reasoning, deductive databases and logic programming with negation as failure. Since then, it has been considered a primary candidate as an effective knowledge representation tool.

For the scope of this project, we use the *clingo* ASP system, which is a combination of the grounder *gringo* and the solver *clasp*, developed by the Potassco research group from the University of Potsdam (Gebser et al., 2014). This system provides a useful API for *python* which allows us to integrate our different approaches.

2.4 Inductive Learning of Answer Set Programs (ILASP)

According to Law et al. (2015):

> "The goal of Inductive Logic Programming (ILP) is to find a hypothesis that explains a set of examples in the context of some pre-existing background knowledge."

Inductive Logic Programming can be used in many applications. In the context of learning how to play games, the hypotheses we want to learn will correspond to the strategy. The background knowledge will be the rules of the game and the set of observations are the most desirable behavior in a given context. One of the advantages from using ILP over statistical machine learning approaches is that the learned hypotheses can be easily expressed in plain English and explained to a human user.

In this project, we use the Inductive Learning of Answer Set Programs (ILASP) framework, which is an inductive logic programming framework developed largely by Mark Law from the Imperial College London. It uses the system *clingo* to generate answer sets explaining the examples. The framework can also generate *weak constraints* as hypotheses using *ordered examples* (Law et al., 2015). This particular feature allows us to find a game strategy in the form of weak constraints given an ordered set of preferred moves in a given context. Such characteristics make it a great fit to learn strategies for games defined in ASP.

Example 3.2. Consider the learning task:

```
animal(cat).    animal(dog).    animal(fish).    eats(cat, fish).    eats(dog, cat).
0 { own(A) } 1 :- animal(A).

#pos(p1, { own(fish) }, { own(cat) }).              #pos(p2, { own(cat) }, { own(dog), own(fish)}).
#pos(p3, { own(dog), own(cat), own(fish) }, {}).    #pos(p4, { own(dog)}, { own(fish) }).
#pos(best, { own(dog), own(fish) }, { own(cat) }).
                                                     % These ordering examples all use an example "best" in
                                                     % which a dog and a fish is owned and a cat is not owned.
                                                     %
#brave_ordering(best, p1).                           % "best" is better than some situations where we
                                                     % own a fish but no cat.
#brave_ordering(best, p2).                           % "best" is better than some situations where we
                                                     % own a cat but no fish or dog.
#brave_ordering(best, p3).                           % "best" is better than some situations where we
                                                     % own a fish, dog and cat.
#cautious_ordering(best, p4).                        % "best" is better than every situation in which
                                                     % we own a dog but no fish.
```

Fig. 3. Sample ILASP program; excerpt from ILASP v3.1.0 manual in Law et al. (2017)

2.4.1 Definitions and Syntaxes

Weak constraint

A weak constraint represents an expression of preference in ASP; such that resulting answer sets of a program P can be ranked by preference. Therefore, a weak constraint represents a form of optimization in ASP.

Partial interpretation

A partial interpretation E is a pair of sets of atoms E^{inc} and E^{exc}, called the inclusions and exclusions of E. We write E as $\langle E^{inc}, E^{exc} \rangle$. An answer set A is said to extend E if it contains all of the inclusions ($E^{inc} \subseteq A$) and none of the exclusions ($E^{exc} \cap A = \emptyset$).

Positive/Negative example

A positive/negative example is a partial interpretation and can be specified in ILASP as `#pos(example_name,`$\{e_1^{inc}, \ldots, e_m^{inc}\}, \{e_1^{exc}, \ldots, e_n^{exc}\}$`)` or `#neg(example_name,`$\{e_1^{inc}, \ldots, e_m^{inc}\}, \{e_1^{exc}, \ldots, e_n^{exc}\}$`)` respectively, where `example_name` is a unique identifier for the example.

Language bias

A language bias in ILASP refers to the search space of possible hypotheses in which inductive solutions should be searched for. These are typically encoded by mode declarations. An example of this is `#modeo`, which expresses what is allowed to appear in the body of a learned weak constraint.

Ordering example

An ordering example is a tuple $o = \langle e_1, e_2 \rangle$, where e_1 and e_2 are partial (positive-example) interpretations.

Brave ordering[1]

An ASP program P bravely respects o iff $\exists A_1, A_2 \in AS(P)$, such that $AS(P)$ is the answer set of P, A_1 extends e_1, A_2 extends e_2 and $A_1 \succ_p A_2$.

Cautious ordering[1]

An ASP program P cautiously respects o iff $\forall A_1, A_2 \in AS(P)$, such that $AS(P)$ is the answer set of P, A_1 extends e_1, A_2 extends e_2 and $A_1 \succ_p A_2$.

2.4.2 ILASP Framework

Law et al. (2017) provides a succinct overview of the ILASP framework to inductively learn hypotheses from answer sets:

> Given an ASP program B called the background knowledge, a set of ASP rules S called the search space and two sets of partial interpretations E^+ and E^- called the positive and negative examples respectively, the goal is to find another program H called a hypothesis such that:
>
> 1. H is composed of the rules in S ($H \subseteq S$)
> 2. Each positive example is extended by at least one answer set of $B \cup H$ (can be a different answer set for each positive example)
> 3. No negative example is extended by any answer set of $B \cup H$

Similarly, Law et al. (2017) provides a framework for extending the above-mentioned method to learn weak constraints from positive examples and ordered answer sets. For brevity, we will exclude the full details of this process; but it is similar to the learning process defined above. For the scope of this project, we will use ordering examples to inductively learn weak constraints. As a result, we will only consider positive examples as partial interpretations.

2.5 Game Tree

In order to learn any game strategies, we would first have to conceptualize the concept of a two-player game in a logical and well-computable format. One such useful format would be a game tree. A game tree is a directed graph whose nodes are states of the game and whose edges are action. The *complete game tree* of a game starts at the initial position and contains all legal actions from each position.

For this project, we represent such trees using the `anytree` Python package, which is publicly available in the Python Package Index (PyPI) repository. Given a game tree T, we assume the following conventions pertaining to tree objects in the scope of this paper:

1. $T(i, j, p, d)$ refers to the node with identifier j present at depth d whose parent node has identifier i at depth $d - 1$, where player p is playing the current round.
2. For cases where no parent node exists, such as for the root node, the following convention will be used: $T(\text{null}, j, p, d)$
3. i, j and d defined above all exist in $\mathbb{N}_0$
4. d has an upper bound of D, where D refers to the maximum depth of tree T. d corresponding to 0 would imply the root node.

[1] **Note:** Given two positive examples e_1 and e_2 with identifiers $\texttt{id}_1$ and $\texttt{id}_2$, the ordering example $\langle e_1, e_2 \rangle$ can be expressed in ILASP as `#brave_ordering(order_name,id1,id2)` or `#cautious_ordering(order_name,id1,id2)`, where `order_name` is an optional identifier for the ordering example.

5. We will assume that only two players play the corresponding games alternately. As a convention, we will define these players as player a and player b. Therefore, p can take the values of a or b.

6. $T(i, j, p, d)[s]$ will refer to the state s of the node $T(i, j, p, d)$. This state contains information on the current configuration of the game and whether the state is terminal.

7. $T(i, j, p, d)[a]$ will refer to the action a that was taken to reach the node $T(i, j, p, d)$. Only the root node will have no associated action.

8. $T(i, j, p, d)[v]$ will refer to the reward value of the node $T(i, j, p, d)$. This will usually only be defined for terminal nodes of a game tree.

9. $\mathcal{M}(s_0, a_0, \cdots, a_n - 1, s_n)$ will refer to a match in the game as a sequence of states. A match can be obtained form a path starting in root node and traversing the tree util a leaf node is reached.

10. Given a match $\mathcal{M}_a(s_0, a_0, \cdots, a_n - 1, s_n)$ we can generate a tree T_a with a single branch rooted in state s_0.

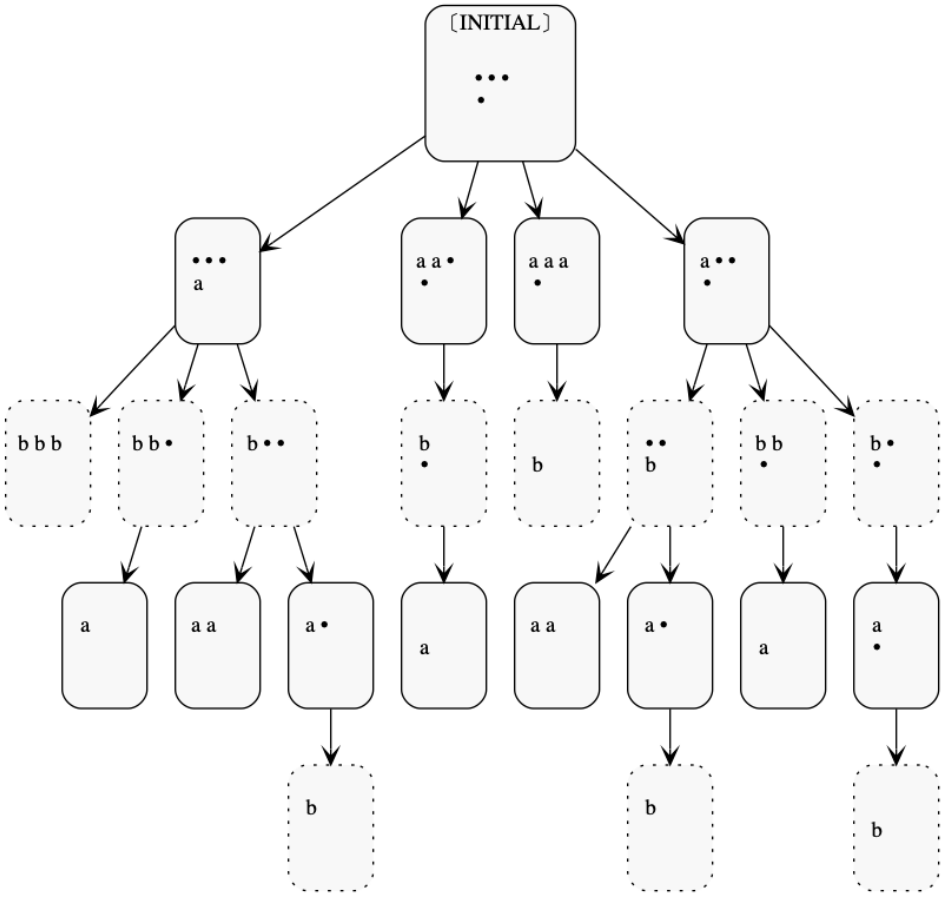

Fig. 4. Complete game tree for Nim with initial configuration (3,1,0,0). The actions are represented as part of the nodes instead of the edges.

3 Methodologies

3.1 Game simulation

To simulate the entirety of a game, we utilized a combined ASP and Python framework, which is summarized in a later subsection. In regards to ASP, we created an encoding, namely `background.lp`, representing the dynamics of the game to perform a single time step. This encoding depends on a set of facts defining the current state of the game. For the initial state, such facts are saved in a file and for the following states, they are computed throughout the simulation. Initial states are saved in files `*initial.lp`, where * implies a prefix. Both sets of encodings are written using the GDL formalisms, such that the same variable and atom naming conventions can be used across all games. For the game of Nim illustrated in Figure 4, the background program and initial state can be found in Figure 5 and Figure 6, respectively.

```
1  #const max_removable = 6.
2  #const max_sticks = 7.
3  #const num_piles = 4.
4
5  % Roles
6  role(a). role(b).
7
8  % Base (All possible values inside 'true')
9  base(has(P,M)):- P=1..num_piles,M=0..max_sticks.
10 base(control(X)) :- role(X).
11
12 % Input (All possible actions)
13 input(X,remove(P,N)) :- P=1..num_piles,
14                        N=1..max_removable,
15                        role(X).
16
17 % Legal actions
18 legal(X,remove(P,N)) :- true(has(P,M)),
19                        P = #min{L:true(has(L,M))},
20                        N=1..max_removable, N<=M,
21                        true(control(X)),
22                        not terminal.
23
24 % Action selection
25 0 {does(X,A)} 1 :- legal(X,A), not terminal.
26 :- does(X,Y), does(X,Z), Y < Z.
27 :- not does(X,_), true(control(X)), not terminal.
28
29 % State transition
30 next(control(b)) :- true(control(a)), not terminal.
31 next(control(a)) :- true(control(b)), not terminal.
32 next(has(P,N-M)) :- does(_,remove(P,M)),
33                     true(has(P,N)), not terminal.
34 next(has(P,N)) :- not does(_,remove(P,_)),
35                   true(has(P,N)), not terminal.
36
37 % Goals
38 goal(X,-1) :- #sum{N,M:true(has(M,N))}=0, true(control(X)).
39 goal(X,-1*G):- goal(Y,G), role(X), X!=Y.
```

```
40
41 % Terminal state
42 terminal :- goal(_,_).
```

Fig. 5. `background.lp` encoding for dynamics of the game of Nim

```
1 true(has(1,3)).
2 true(has(2,1)).
3 true(has(3,0)).
4 true(has(4,0)).
5 true(control(a)).
```

Fig. 6. ASP encoding the initial state of Figure 4

Throughout the game simulation, each node $T(i, j, p, d)$ of the tree can expanded with the following actions. Using the background program P and the set of facts representing the state $T(i, j, p, d)[s]$, we call *clingo* in search for the next possible states. This call will obtain one answer set per legal action. For each answer set, we proceed to create a new children node $T(i, j', p', d + 1)$ were its state $T(i, j', p', d + 1)[s]$ is constructed by replacing `next/1` with predicate `true/1`. To exemplify such process, show the output when we call *clingo* with the encodings from Figure 5 and Figure 6. This call will return 4 answer sets corresponding to the first level of the tree in Figure 4.

```
Answer: 1
next(has(3,0)) next(has(4,0)) next(control(b))
does(a,remove(2,1)) next(has(1,3)) next(has(2,0))

Answer: 2
next(has(3,0)) next(has(4,0)) next(control(b))
does(a,remove(1,2)) next(has(2,1)) next(has(1,1))

Answer: 3
next(has(3,0)) next(has(4,0)) next(control(b))
does(a,remove(1,3)) next(has(2,1)) next(has(1,0))

Answer: 4
next(has(3,0)) next(has(4,0)) next(control(b))
does(a,remove(1,1) next(has(2,1)) next(has(1,2))
```

Using the Python Application Programming Interface (API) for *clingo*, we can then extrapolate this process to simulate multiple time steps in the game and construct the game tree. Further, when given a strategy in the form of weak constraints, such encoding can be included, enforcing an order over the answer sets. Such ordering will allow the player you select the best action (answer set) according to the optimization defined by the weak constraints.

3.2 Learning Approaches

3.2.1 Minimax Algorithm

The minimax algorithm is a recursive tree-search algorithm which has its origins in combinatorial game theory for choosing the next best move of a two-person zero-sum game. The intuition behind this algorithm lays in the fact that every movement made by player a will aim to maximize the reward of a, while player b will try to minimize this same reward. This algorithm will first construct the complete game tree T of depth D for alternately-playing players p and p'. All the terminal nodes of the tree will be annotated with reward values v extracted from predicate `goal/1`. Then, it will proceed to compute the minimax score M of any node on the tree starting on the leaves by utilizing the following piecewise function:

$$M(j, p, d-1) = \begin{cases} \max\limits_{k} T(j, k, p', d)[v], & p = a \\ \min\limits_{k} T(j, k, p', d)[v], & p = b \end{cases} \tag{1}$$

where:

d = depth of node on the decision tree: $d \in [0, D],\ d \in \mathbb{N}_0$
j = j'th parent node on depth $d-1$ where M is to be evaluated: $j \in \mathbb{N}_0$
k = k'th child node of parent node j on depth d where v is defined: $k \in \mathbb{N}_0$
p, p' = unique player playing on specified depth: $p, p' \in \mathcal{P} = \{a, b\},\ p' = \mathcal{P} \setminus p$
Note: We assume the positive reward player a maximizes M while the negative reward player b minimizes M

Given the piecewise function M defined above, we present below the pseudocode used for the minimax algorithm in our project. The time complexity of this algorithm is $O(b^D)$, while the space complexity is $O(bD)$; where b is the average branching factor and D is the maximum depth of the game tree (Russell and Norvig, 2016).

Algorithm 1: Minimax

Input: Game tree T of depth D with reward values v at terminal nodes
Output: Game tree T' with minimax values at all nodes

```
1: function MINIMAX(T)
2:     T' ← T                                  ▷ Make local copy of T as T'
3:     for d ← D − 1 to 0 do                   ▷ Loop upwards from bottom of T'
4:         for each node T(i, j, p, d) do      ▷ Loop each node at depth d
5:             T'(i, j, p, d)[v] ← M(j, p, d)  ▷ Update T' with M
6:         end for
7:     end for
8:     return T'                               ▷ Return updated game tree T'
9: end function
```

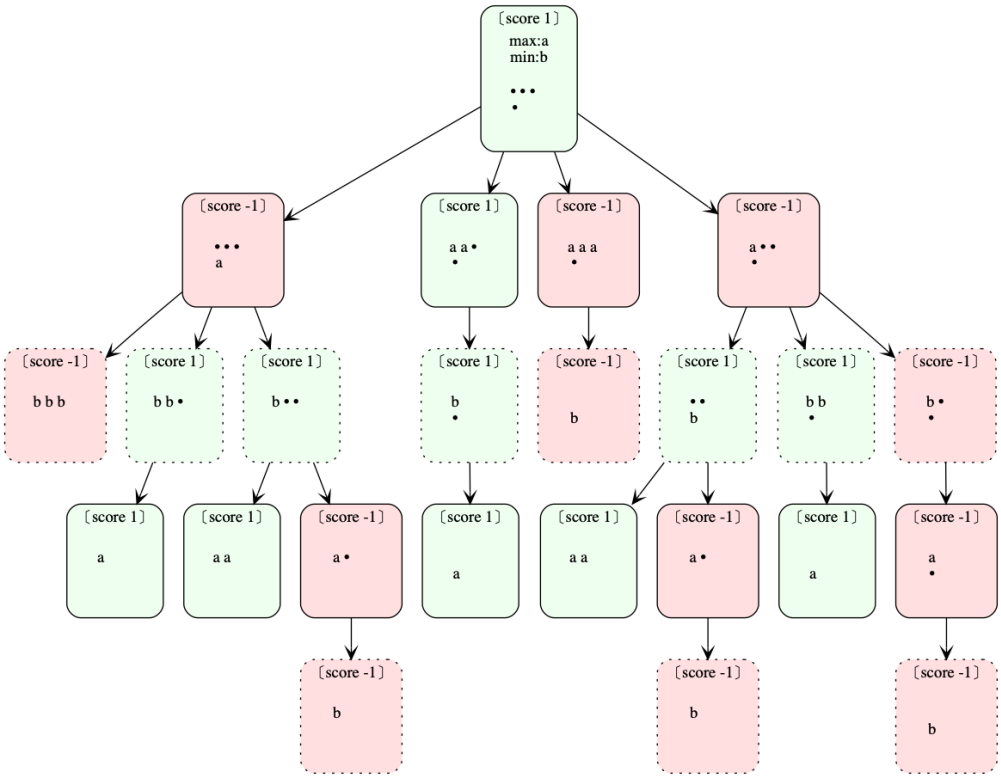

Fig. 7. Tree resulting from applying the Mimimax algorithm to the complete tree in Figure 4. Green nodes have a positive score, implying a winning strategy for player a, while red nodes represent an position where player b will win.

In the example given on Figure 7, all nodes are scored using the minimax function M, defined in formula 1. Since the score of the root node is 1, player a can assure a winning outcome by following the path down the tree choosing the nodes that have this score, such nodes are marked in green. We can notice this path is the same one as the one defined on Figure 1.

3.2.2 Pruned Minimax Algorithm

The Minimax algorithm has many variants that aim to make it more efficient by decreasing the number of nodes that are evaluated. One of the best-known algorithms, called Alphabeta pruning (Knuth and Moore, 1975), stops evaluating a move when at least one possibility has been found that proves the move to be worse than a previously examined move. Our proposed Pruned Minimax algorithm is based on this idea while including the optimization statements existent in *clingo*. The construction is performed similarly to the Depth-first search (DFS) algorithm for traversing or searching tree.

Explicit time encoding

In GDL we use predicate `true(F)` to indicate that `F` holds in the current state and `next(F)` when it holds in the next state. This syntax can be translated to a single predicate `holds(F,T)` where the time step in which `F` holds is specified with `T`. This notation is used for planning problems and it is based on event calculus Shanahan (1999). With the new encoding, each stable model will represent a full match containing the actions performed in each time step. This representation

makes use of all the advantages provided by *clingo* for planning problems such as incremental solving. In order to translate our current encoding to this new format we need to perform the following steps:

1. Replace `true(F)` by `holds(F,T)`
2. Replace `next(F)` by `holds(F,T+1)`
3. Replace `p(F1..Fn)` by `p(F1..Fn,T)` where $p \in \{legal, does, goal, terminal\}$
4. In every rule where a replacement was made, add the time in in the body with predicate `time(T)`.
5. Add a new fact with all possible time steps `time(0..N)` using a predefined horizon N.

Optimization

The system *clingo* comes with a set of optimization statements that enforce an order in the answer sets. With the optimization statement `#maximize{N,T:goal(a,N,T)}` we can find the stable model which maximizes the reward N given to player a defined in the predicate `goal(a,N,T)`, were T is the time step in which the goal was reached. By adding this to the explicit time encoding of a game, we will find the match that maximizes the reward for a. However, this optimization assumes that player b will make the moves that generate the best outcome for his opponent. We will call this match $\mathcal{M}_a$ an optimized match for a. To select actions for player b working on his own benefit, we would need to minimize the reward of a in the time steps where player b has control. Minimizing the reward of a can be achieved with the optimization statement `#minimize{N,T:goal(a,N,T)}`. Sadly, the alternation of different optimizations in every time step is not allowed in *clingo*. Nonetheless, we can perform such alternation with multiple calls to the solver in a clever manner as it is described bellow.

The algorithm

The algorithm in this approach uses the defined explicit time encoding to find optimal matches (stable models) and proceeds to modify the matches such that they are optimizing both players. This process, which we will call *validation*, will construct a scored-pruned game tree leaving irrelevant parts of the tree unexplored. The pseudocode for the algorithm can be found below in Algorithm 2 and 3. The recursive function *validate* will perform a bottom-up validation from the the end of the match up until step i. Lines 3 and 4 make sure that the rest of the match is fixed and a new part is explored by negating the action already taken at each step. Lines 14 and 15 append new paths to the game tree. The pruning is performed on lines 19 and 21 where the current branch of the tree remains only explored by the clingo call (Line 7 or 9). Validation of this branch unnecessary since the current result can't be improved[2]. The recursive call in line 23 will proceed to explore the

[2]These unexplored branches can be found in Figure 8 with blank nodes.

required parts of the tree by validating the match. Finally, the case on line 29 will update the match and the score of the node with an improved action for the player in turn.

Algorithm 2: Pruned minimax

Input: The explicit encoding P_e and the initial state of the game
Output: A tree T for the pruned tree

1: **function** PRUNEDMINIMAX(P_e,s_0)
2: $\mathcal{M}_a(s_0, a_0, \cdots, an-1, s_n) \leftarrow clingo(P_e, s_0)$ ▷ Compute $\mathcal{M}_a$
3: $T \leftarrow [s_0, \cdot, s_n]$ ▷ Create a tree with only the branch from $\mathcal{M}_a$
4: $M_v \leftarrow validate(M_a, 0)$
5: **end function**

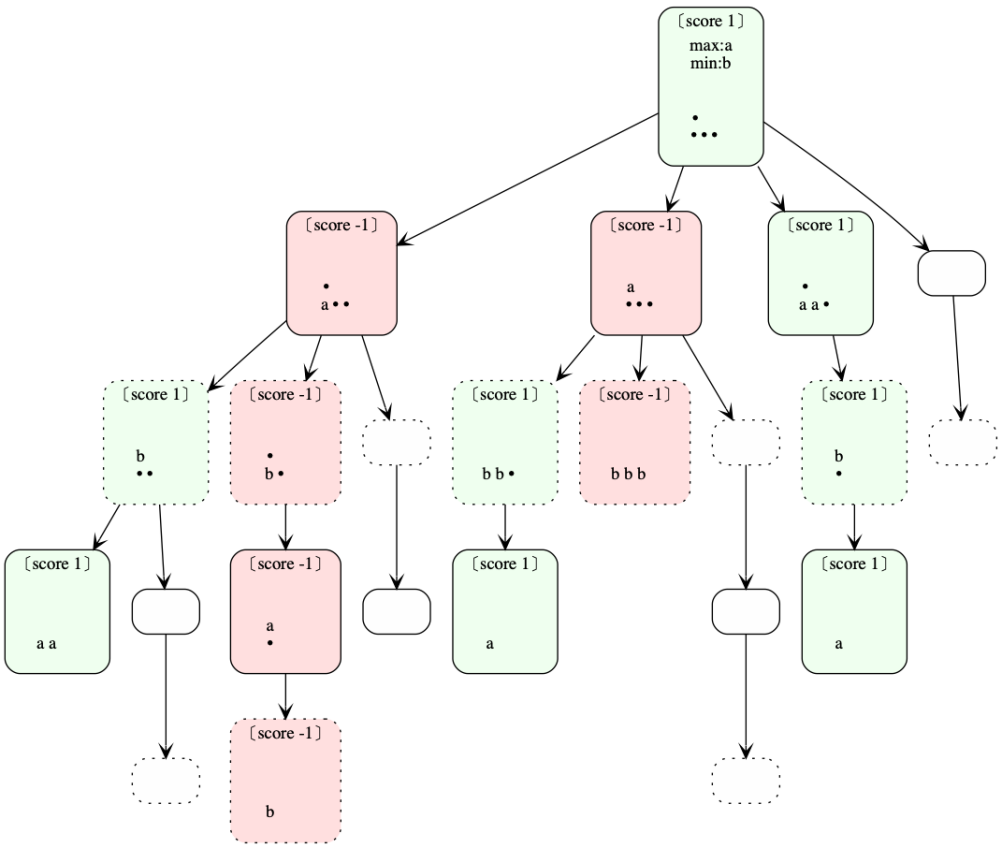

Fig. 8. Tree resulting from applying the Pruned Minimax algorithm to the complete tree in Figure4. White branches are those that remain only explored by clingo but never validated.

Learning ASP rules

During the validation process, there are some key points where we become aware of one action being better than another. This points are on lines 18, 25 and 29 of Algorithm 2 where a player should take a specific action to achieve a better reward. It is in these sections where we can generate rules that define a strategy or other type of data as explained in the next subsection. This strategy

Algorithm 3: Validate match

Input: An optimized match M_v to be validated from a time step i
Output: A validated match where both players play to their advantage

function VALIDATE($\mathcal{M}_v(s_0, a_0, \cdots, s_i, a_i, \cdots, s_n)$,$i$)
  for $k \leftarrow n-1$ to i **do** ▷ Loop upwards each state untill s_i
    $P_k \leftarrow [s_0, a_0, \cdots, s_k]$ ▷ Fix states and actions untill k
    $P_k \cup not\ a_k$ ▷ Negate action a_k to explore other possible actions
    $p_c \leftarrow$ player in control on s_k
    if $a == p_c$ **then**
      $M_{p_c} \leftarrow clingo(P_e, P_k, max)$ ▷ Compute match from k maximizing a
    else
      $M_{p_c} \leftarrow clingo(P_e, P_k, min)$ ▷ Compute match from k minimizing a
    end if
    if M_{p_c} is $None$ **then** ▷ No other possible action than a_k
      continue ▷ Go to next k
    end if
    $T_{p_c} \leftarrow [s_{p_c}k+1, \cdots, s_{p_c}m]$ ▷ Create a tree with only the branch from M_{p_c}
    Hang unexplored branch T_{p_c} from the parent of s_k in T
    $r_o \leftarrow M_{p_c}[p_c]$ ▷ Reward for p_c in optimized
    $r_v \leftarrow M_v[p_c]$ ▷ Reward for p_c in old match
    if $r_v > r_o$ **then** ▷ a_k is the best action for p_c
      $T(x, s_k, k, p_c)[v] \leftarrow r_v$
    else if $r_v == r_o$ **then** ▷ a_k is as good as the best outcome for p_c
      $T(x, s_k, k, p_c)[v] \leftarrow r_v$
    else if $r_v < r_o$ **then** ▷ Action in M_{p_c} potentially better than a_k for p_c
      $M_{p_c v} \leftarrow validate(M_{p_c}, k+1)$ ▷ Validates the match and explores T_{p_c}
      $r_{p_c v} \leftarrow M_{p_c v}[p_c]$ ▷ Reward for p_c in validated match
      if $r_v > r_{p_c v}$ **then** ▷ a_k is the best action for p_c
        $T(x, s_k, k, p_c)[v] \leftarrow r_v$
      else if $r_v == r_{p_c v}$ **then** ▷ a_k is as good as the best outcome for p_c
        $T(x, s_k, k, p_c)[v] \leftarrow r_v$
      else if $r_v < r_{p_c v}$ **then** ▷ The action in $M_{p_c v}$ is better for p_c
        $M_v \leftarrow M_{p_c v}$
        $T(x, s_k, k, p_c)[v] \leftarrow r_{p_c v}$
      end if
    end if
  end for
  return M_v ▷ Return match M_v validted from step i
end function

can be considered an abstract representation of the scored game tree. Within the Algorithm 2 we can easily use the current information to generate rules of the form:

```
best_do(a,remove(2,2),T):-  holds(control(a),T),
                            holds(has(1,0),T),
                            holds(has(2,2),T),
                            holds(has(3,0),T),
                            holds(has(4,0),T).
```
(2)

These rules will define under which state context, described by predicate `holds/2` is it better to perform a certain action. In this case, for the game on Nim we are stating that: In a state where player a has control and only pile 2 has 2 counters it would be best to remove from pile 2 both counters.

It is important to notice that the rules in this approach will only be correct when the body defines one unique state of the game. This will imply a new requirement that must be fulfilled by the background encoding for this approach to be correct.

In order to enforce the use of these preferred actions as an strategy all we need to do is add the following rule to the strategy.

```
1{does(P,A,T):best_do(P,A,T)}1:- time(T),
                                 not goal(_,_,T),
                                 {best_do(P,A,T)}>0,
                                 true(control(P)).
```
(3)

A desirable characteristic of a strategy is to be generalizable. Since the generated rules use constants, this can't be achieved. However, given a formalization allowing us to decide whether a term should be converted into a variable or not, we can substitute each some constants by variables. For this formalization we used a function $f : P, i \rightarrow Bool$ where every function name in P has an associated boolean value for each attribute position i indicating if it should be transformed into a variable. We will also need additional literals to ensure different variables are not substituted by the same constant during grounding. The following rule would correspond to the previous example generalized with variables using only $f(has, 0)$,$f(control, 0)$ and $f(best\_do, 0)$ are assigned to true.

```
best_do(Va,remove(V2,1),T):-  holds(control(Va),T),
                              holds(has(V1,0),T),
                              holds(has(V2,2),T),
                              holds(has(V3,0),T),
                              holds(has(V4,0),T),
                              V1!=V3,V1!=V2,V1!=Va,
                              V3!=V2,V3!=Va,V2!=Va.
```
(4)

With this generalize rule we can apply the strategy to any permutation over the piles and players. Further, by applying these rules during the construction of the tree we can prune more branches with the learned information. This is exemplified in figures 9 and 10. In Figure 10 we can notice how the right branch of the tree is never explored. This is because during the construction of the left branch in the bottom left corner we learned the rule mentioned above. When this rule is generalized it becomes applicable for player b. Since player b now knows that removing two counters is better, the green section of the right branch from Figure 9 is removed leaving the only possible outcome of -1 and becoming unnecessary to be further explored by a.

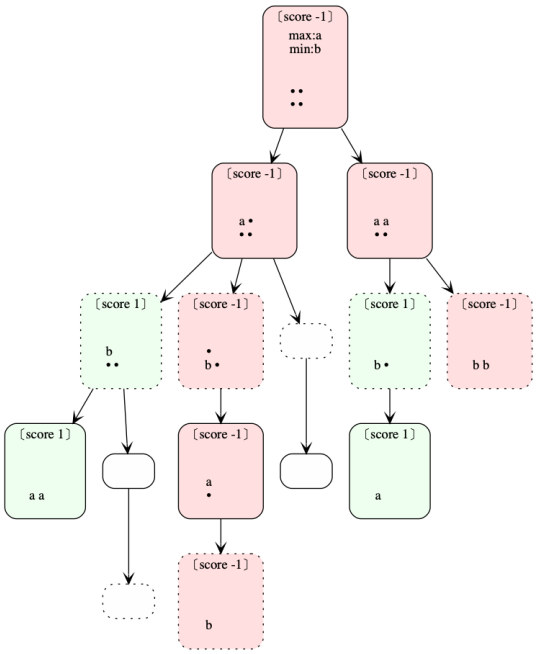

Fig. 9. Pruned minimax tree without applying learned rules.

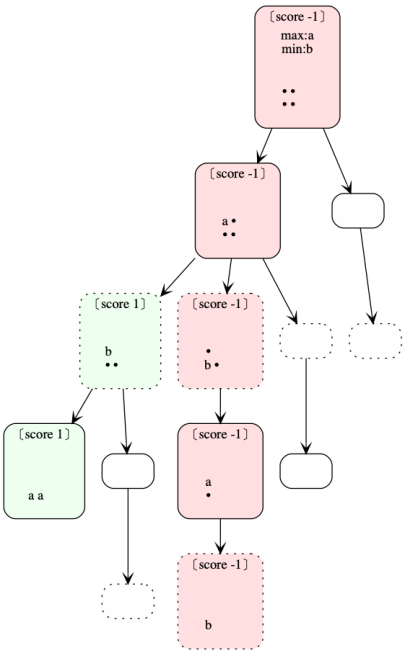

Fig. 10. Pruned minimax tree applying learned rules.

3.2.3 Weak Constraints from ILASP

For this approach, our objective is to find more generalized strategies in the form of weak constraints using ILASP. Since we already have the background knowledge defined, we just need to define a set of ordered positive examples and a language bias.

Generating ordered examples

The decisive points mentioned in the previous section to learn rules can also be used to generate ordered examples. By running the Pruned Minimax algorithm we can construct examples with small modifications to the code. The ordered example corresponding to rule in Example 2 would be the following:

```
#pos(e0,{}, {}, {
    true(control(a)). true(has(1,0)). true(has(2,2)).
    true(has(3,0)). true(has(3,0)). does(a,remove(2,2)).
}).
#pos(e1,{}, {}, {
    true(control(a)). true(has(1,0)). true(has(2,2)).
    true(has(3,0)). true(has(4,0)). does(a,remove(2,1)).
}).

#brave_ordering(e0,e1).
```

Defining language bias

Once we have the examples we only need to define a language bias. This process in most cases will require time to define how the strategy (Hypothesis) might look. The code in Figure 11 shows the language bias used for the game if TicTacToe. Line 1 to 4 define the possible atoms that might appear in the week constraint. These atoms include a cell being free, tree cells being on a line and the values for each cell. It is important to notice the use of `next/1` instead of `true/1`. Creating a strategy dependent on the next state will automatically include the effect of the action taken, thus indirectly ordering the actions. Lines 5 to 8 define the configuration for the possible weights, priorities and variables.

```
1 #modeo(3,next(has(var(player),var(cell))),(positive)).
2 #modeo(1,in_line(var(cell),var(cell),var(cell)),(positive)).
3 #modeo(1,next(free(var(cell))),(positive)).
4 #modeo(1,next(control(var(player))),(positive)).
5 #weight(-1).
6 #weight(1).
7 #maxp(2).
8 #maxv(4).
```

Fig. 11. Language bias for the game of TicTacToe

For the game of Nim we wanted to find a strategy close to the mathematical one defined in Nim. To do so, we first included in the encoding a set of facts for the binary representation of a number. Atom `binary(N,D,V)` can be read as number N has the value $V \in \{1, 0\}$ in position $D \in \{1, 2, 3\}$. Since the expected strategy is more complex and by using the existent predicates the search space is too big, we used the language bias for predicate invention. For the invention, ILASP will generate normal ASP rules in addition to the weak constraints. In the code on Figure 12, lines 5 to 7 define the invented predicates that will go in the head of the ASP rules where the body will be defined by lines 8 to 11. Lines 12 and 13 describe the possible predicates for the weak constraints.

```
#constant(pile,1).
#constant(pile,2).
#constant(pile,3).
#constant(pile,4).
#modeh(b(var(pile),var(d),var(bool)),(positive)).
#modeh(nim_sum(var(d),var(total)+var(bool),var(pile)),(positive)).
#modeh(nim_sum(var(d),0,0),(positive)).
#modeb(1,b(var(pile),var(d),var(bool)),(positive)).
#modeb(1,nim_sum(var(d),var(total),var(pile)-1),(positive)).
#modeb(1,binary(var(num),var(d),var(bool)),(positive)).
#modeb(1,next(has(var(pile),var(num))),(positive)).
#modeo(1,nim_sum(var(d),var(t),const(pile))).
#modeo(1,var(t)\2 != 0).
#weight(1).
#weight(-1).
#maxp(1).
#maxv(4).
```

Fig. 12. Language bias for the game of Nim

As a final step, we will use the generated ordered examples, the language bias and the background encoding to call the ILASP framework. The call will return a hypothesis representing a strategy that will order the examples. If no Hypothesis is found the result will be UNSAT.

3.3 Workflow Summary

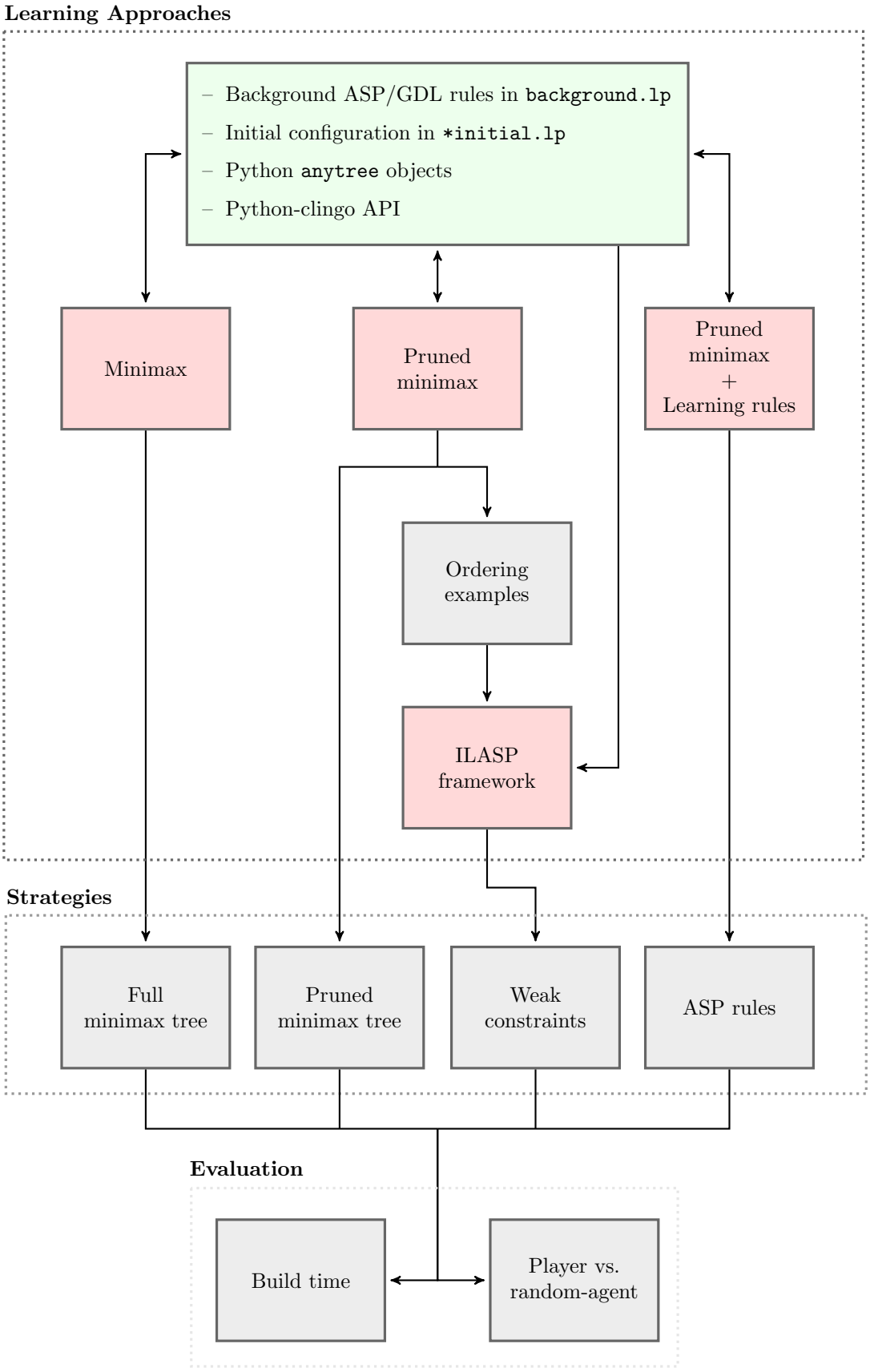

Fig. 13. Summary of workflow/methodologies in this project separated into three different categories; namely learning approaches, strategies and evaluation

4 Results

We divided the evaluation into two phases. In the first one, we examined the process of learning each strategy and the results. For the second one, we tested their performance during gameplay. The presented results ran on a 2.6 GHz Quad-Core Intel Core i7 processor under MacOs. The benchmarks were performed using three different initial configurations for each game of increasing complexity, namely, *Small(S), Medium(M)* and *Large(L)*. These divisions were designed to provide information about the scalability of each approach into the *Classical* initial setting.

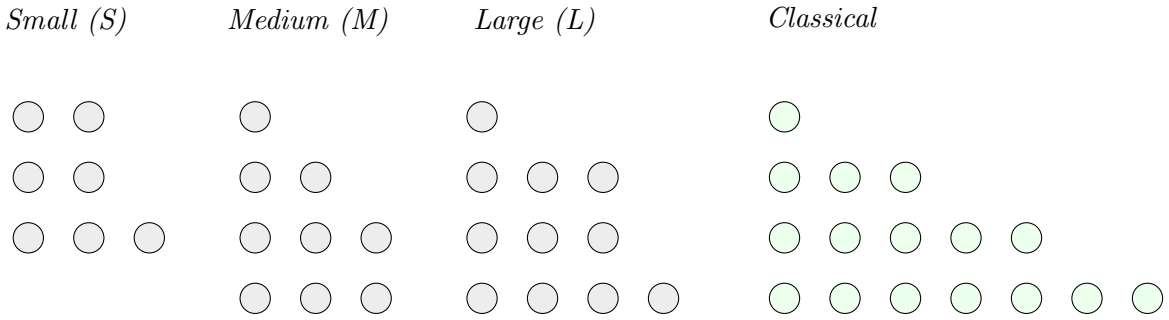

Fig. 14. Initial configurations for *Nim*.

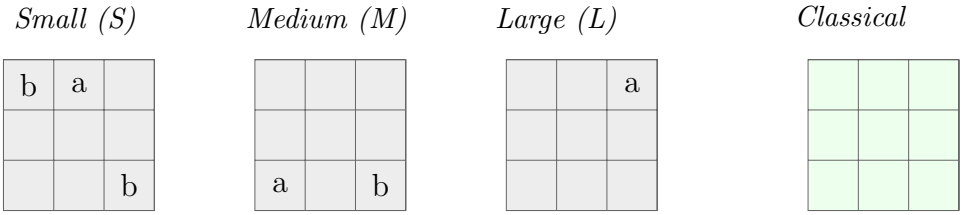

Fig. 15. Initial configurations for *TicTacToe*

4.1 Learning phase

The learning phase of the evaluation consisted of computing the strategy for each approach. The strategies were computed for each initial state and saved for the next phase. We will proceed to interpret different aspects of the this process as well as the resulted strategies.

Approach	S	M	L
Minimax	390	4630	61023
Pruned minimax tree	83	347	2675
Pruned minimax rules	35	85	283

Table 1. Number of nodes in the computed tree for Nim

Approach	S	M	L
Minimax	896	7583	59704
Pruned minimax tree	105	73	2835
Pruned minimax rules	109	61	2399

Table 2. Number of nodes in the computed tree for TicTacToe

The trees computed for the Minimax and the Pruned Minimax approaches are too large to be presented in this paper. Nevertheless, we can examine their structure by looking at the number of nodes in each tree. Such numbers are included in Figure 16 and their detailed values can be found in Tables 1 and 2. From those tables, we can notice the significant reduction in the tree size when using the pruning strategy compared to the full tree. As well as a further reduction by including the learned rules in the learning.

Unlike game trees, the weak constraints generated by the ILASP approach for each initial instance are concise and explainable. The strategy found for Nim (Figure 16), encode the mathematical strategy explained along with the game description. Lines 1, 2 and 3 define new predicates `nim_sum/3` and `b/3` to save the *Nim Sum* and the binary number of each digit for every pile, respectively. Using this new predicates, the weak constraint in line 4 penalizes an odd *Nim Sum* for any digit. The strategy learned using the small instance only varies by penalizing *nim_sum(V0,V1,3)* instead of *nim_sum(V0,V1,4)*, this change is associated to the lack of counters on pile 4 for the small instance.

The strategy found for TicTacToe with initial state M (Figure 18) will give preference to stable models where there is a free space in a consecutive line, stated in line 1. A higher priority of 2 will be assigned to stable models where player $V0$, corresponding to the current player, completes a full line in the next state. This last fact encoded in lines 2 and 3, corresponds to the expected strategy of always choosing the action leading to a winning state. In the case of the initial state S (Figure 17), lines 3 and 4 will penalize actions where the next state has a line with one cell taken and one free. Thus, encouraging actions that block the other player moves.

```
1 nim_sum(V1,0,0) :- b(_,V1,_).
2 nim_sum(V1,V3+V2,V0) :- b(V0,V1,V2), nim_sum(V1,V3,V0-1).
3 b(V3,V1,V2) :- binary(V0,V1,V2), next(has(V3,V0)).
4 :~ nim_sum(V0,V1,4), V1\2 != 0.[1@1, 4, V0, V1]
```

Fig. 16. Strategy found by ILASP for Nim instances M and L

```
1 :~ next(has(V0,V1)), in_line(V1,V2,V3).[1@1, 1, V0, V1, V2, V3]
2 :~ in_line(V0,V1,V2), next(free(V0)).[-1@2, 2, V0, V1, V2]
3 :~ next(has(V0,V1)), in_line(V2,V3,V1),
4    next(free(V2)).[1@2, 3, V0, V1, V2, V3]
5 :~ next(has(V0,V1)), next(has(V0,V2)),
6    next(has(V0,V3)), in_line(V1,V2,V3).[-1@2, 4, V0, V1, V2, V3]
```

Fig. 17. Strategy found by ILASP for TicTacToe instance S

```
1 :~ in_line(V0,V1,V2), next(free(V2)).[-1@1, 1, V0, V1, V2]
2 :~ next(has(V0,V1)), next(has(V0,V2)),
3    next(has(V0,V3)), in_line(V1,V2,V3).[-1@2, 2, V0, V1, V2, V3]
```

Fig. 18. Strategy found by ILASP for TicTacToe instance M

We present in Figure 16 the time taken by each approach to generate the strategy. These times are taking into account the saving process for each strategy,

in the case of strategies in the form of trees, this process exported the tree into JSON format to provide faster access. The building times for ILASP include the generation of ordered examples from the Pruned Minimax approach as well as the ILASP process to compute the Hypothesis.

We notice how for the small TicTacToe the times for ILASP are considerably bigger than the rest. This overload is due to the abstraction process of ILASP, while the rest of the approaches can quickly compute a strategy for a small instance. For the medium size, we can notice that the Minimax approach takes the most time while Pruned Minimax learning rules takes the least time. This behavior correlates with the number of explored nodes. For the larger instance, there are some key points to notice. First, we can see how the game of Nim behaves as it does for the medium size with Pruned Minimax, whereas for TicTacToe we can see a drastic change. In the case of ILASP no strategy was found to cover all the ordered examples. Pruned Minimax with rules outperformed the rest of the approaches in almost every configuration, while for the large TicTacToe instance we can see a bad performance.

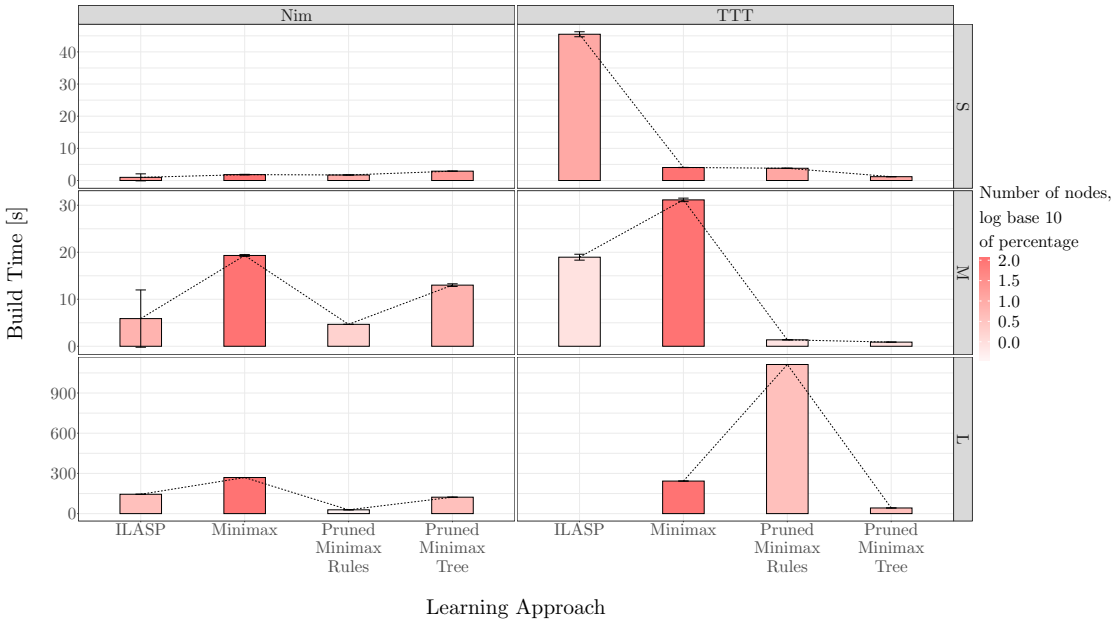

Fig. 19. Learning times for each approach. The bar colors are computed applying log_{10} to the proportion of nodes, with 100 being the tree with the most nodes for each game and size.

4.2 Approaches against Random-Agent

For each learning approach, we created an agent with the capability to choose an action in the current state of the game s_c. This judgment is based on the generated strategies during the learning phase. Table 3 gives a summary of how each agent will select an action during gameplay.

Agent	Strategy	Action selection for current state s_c
Minimax	Full minimax tree	Finds a node in the tree with for state s_c and choose the action corresponding to the children which maximizes the score. If no node is found for s_c, choose a random action.
Pruned minimax tree	Pruned minimax tree	Same as Minimax agent. The unexplored sections of the tree will increase the likelihood of s_c not being part of the tree.
Pruned minimax rules	ASP rules	Includes the rules as part of the clingo call and selects one of the remaining stable models.
ILASP	Weak constraints	Includes the weak constrains as part of the clingo call and selects the optimal stable model.

Table 3. Tabular summary of learned strategies

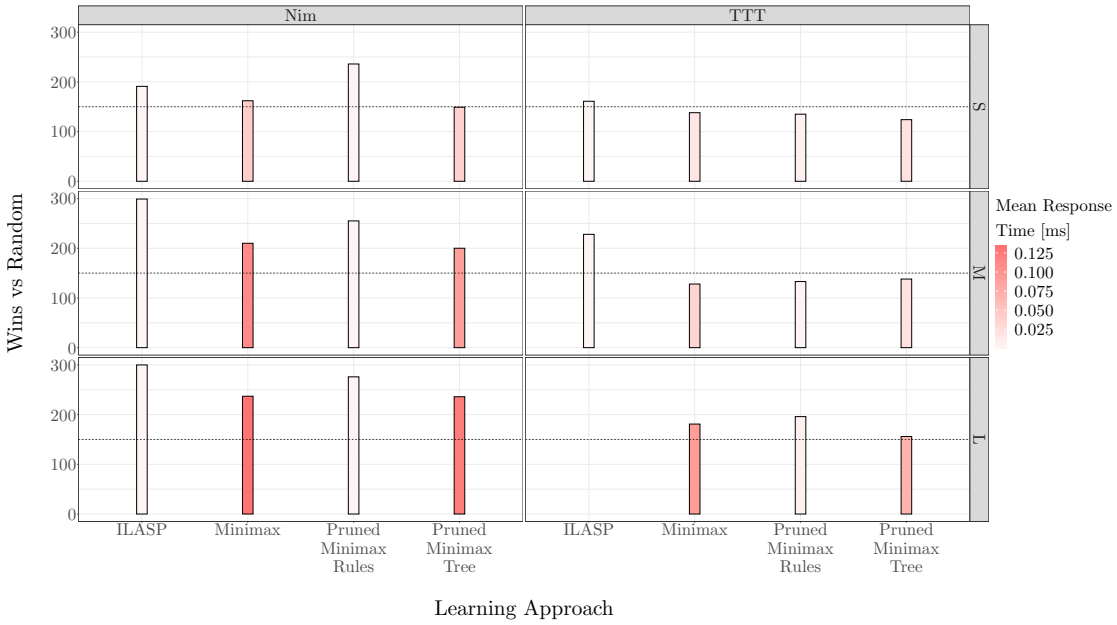

Fig. 20. Number of winning matches for each approach against a random agent including their mean response time.

We played each agent against a Random-Agent and compared the number of wined matches in Figure 20. Using the classical initial state, we simulated a total of 300 matches switching the starting player in each game to avoid an initial advantage to either player.

We can see that out of all approaches that learned from small instances, only the ILASP strategy was able to outperform the random agent in TicTacToe. This behavior continued throughout the experiment showing the potential of ILASP to

generalize into bigger instances. For the game of Nim, the Pruned Minimax agent working with rules was also able to generalize and wan against the random agent in over 75% of the matches. We can also notice a slight improvement when using the full Minimax tree instead of the pruned tree in both games. Regarding the response times, it is easy to notice the increment on response times for indexing the tree. While, on the other hand, adding more information to the encoding showed to provide a fast response.

4.3 Framework

In addition to the research performed on these approaches, we made available a generalized framework. With this framework, we want to encourage the use and research of ASP in two-player games. Our implementation allows simple expansions for new games following the GLD formalisms as well as new approaches for learning strategies. These expansions will automatically generate the command-line tools to build strategies, create agents and compute benchmarks. We also make available the tree visualizations by simply defining an ASCII representation of the states for the new game. The code can be found in github[3], including a detailed explanation of how to extend it with new games and approaches.

5 Discussion

A deeper analysis of the results showed different issues and advantages in each approach taken. Computing the full Minimax tree showed to be generally inefficient. We can see an expected exponential blowup in the number of nodes that correlates with the computation and response times. This outcome was expected as it is related to the brute-force nature of the algorithm. By pruning the tree we notice a clear improvement in the number of nodes. This improvement will also depend on the specific instance and the ordered in with actions are explored. It was due to this random factor that the algorithm was able to skip a big part of the tree in the medium TicTacToe instance, leading to fewer nodes than the ones from the small instance. Even though the pruning had better overall run times and improved space, the contrasting outcome can be seen in the slight difference in performance during gameplay. We can attribute this decrease on winned games, compared to plain Minimax, to the cases where the gameplay falls into the unexplored states of the pruned tree resulting in a random action instead of an informed one.

By learning ASP rules that abstracted the Pruned Minimax tree, we can notice an improvement in the scalability and the number of nodes. This improvement, however, was not the same for both games. By analyzing Figure 20 we can notice a very good performance for Nim, but a poor one for TicTacToe. This can be attributed to the level of abstraction in the rules generated for each game. In TicTacToe strategy rules consist of only constants except for the player's names, whereas the ones presented for Nim, the pile numbers are also substituted by variables. This result can also be seen in the increased build time for the larger instance of TicTacToe. In this case, the number of rules for the strategy increases,

[3]https://github.com/susuhahnml/asp-game-strategies

generating over 1000 rules that are included in each call to *clingo* slowing down its response.

Learning weak constraints with ILASP successfully generalized from small instances into the real game. However, it was not able to reach a conclusive strategy for the large instance of TicTacToe. We noticed that examples provoking this outcome were those referring to earlier stages of the game where a clear strategy is hard to find even for a human; such cases must see over 3 steps into the future to make the decision. The advantage that ILASP provided with explainable and simple strategies can arguably be outweighed by the complexity of the language bias. While the language bias for TicTacToe was intuitive, the one for Nim had to be carefully handcrafted based on previous knowledge of the expected strategy.

6 Conclusions

With this project we were able to learn different strategies for two-player games. We defined a novel variant of the Minimax algorithm which makes use of optimization statements to speed the computation by pruning parts of the game tree. This technique also showed to be good at generalizing for bigger instances when symmetries in the game can be enforced. Even though the strategy it generates as ASP rules can be interpreted by a human, it is not as concise and generalizable as the one learned using ILASP.

Employing Inductive Logic Programming with ILASP system helped us find explainable and scalable strategies. We believe this could be further investigated to tackle some of the downsides of the approach by using of its feature for noisy examples. If we add a weight factor to each example depending on how certain it is of the outcome of the game, we could prioritize strategies in the last stages of the game.

By representing the games with GDL and computing the game dynamics using ASP we created a generalized framework that facilitates the inclusion of new games and techniques for further research. In this project we separated the generation of the strategies for the gameplay. It would also be interesting to explore the option of performing computations during gameplay to improve on the existent strategies.

References

Abu Dalffa, M., Abu-Nasser, B. S., and Abu-Naser, S. S. (2019). Tic-tac-toe learning using artificial neural networks.

Anger, C., Konczak, K., Linke, T., and Schaub, T. (2005). A glimpse of answer set programming. 19:12–.

Bouton, C. L. (1901). Nim, a game with a complete mathematical theory. *The Annals of Mathematics*, 3(1/4):35–39.

Demaine, E. D., Ma, F., and Waingarten, E. (2014). Playing dominoes is hard, except by yourself. In Ferro, A., Luccio, F., and Widmayer, P., editors, *Fun with Algorithms*, pages 137–146, Cham. Springer International Publishing.

Gebser, M., Kaminski, R., Kaufmann, B., and Schaub, T. (2014). Clingo= asp+ control: Preliminary report. *arXiv preprint arXiv:1405.3694*.

Gelfond, M. and Lifschitz, V. (1991). Classical negation in logic programs and disjunctive databases. *New Generation Computing*, 9(3):365–385.

Genesereth, M., Love, N., and Pell, B. (2005). General game playing: Overview of the aaai competition. *AI magazine*, 26(2):62–62.

Kelly, A. R. (2006). One-pile misère nim for three or more players. *International journal of mathematics and mathematical sciences*, 2006.

Knuth, D. E. and Moore, R. W. (1975). An analysis of alpha-beta pruning. *Artificial intelligence*, 6(4):293–326.

Law, M., Russo, A., and Broda, K. (2015). The ILASP system for learning answer set programs. `www.ilasp.com`.

Law, M., Russo, A., and Broda, K. (2017). Inductive learning of answer set programs v3. 1.0.

Lifschitz, V. (2019). *Answer set programming*. Springer International Publishing.

Love, N., Hinrichs, T., Haley, D., Schkufza, E., and Genesereth, M. (2008). General game playing: Game description language specification. *Stanford University*.

Russell, S. J. and Norvig, P. (2016). *Artificial intelligence: a modern approach*. Malaysia; Pearson Education Limited,.

Shanahan, M. (1999). The event calculus explained. In *Artificial intelligence today*, pages 409–430. Springer.

Silver, D., Hubert, T., Schrittwieser, J., Antonoglou, I., Lai, M., Guez, A., Lanctot, M., Sifre, L., Kumaran, D., Graepel, T., Lillicrap, T., Simonyan, K., and Hassabis, D. (2018). A general reinforcement learning algorithm that masters chess, shogi, and go through self-play. *Science*, 362(6419):1140–1144.

Verma, A., Murali, V., Singh, R., Kohli, P., and Chaudhuri, S. (2018). Programmatically interpretable reinforcement learning. *CoRR*, abs/1804.02477.