Learning to Play with Inductive Logic Programming

Graduation Assignment

by

Teun Boschloo

Student number:	8364 25314	
Course code:	IB9906	
Thesis committee:	Dr. ir. Harrie (H.J.M.) Passier (chairman),	Open University
	Dr. ir. Hugo (H.L.) Jonker (supervisor),	Open University
Project sponsor:	Dr. Jesse Heyninck,	Open University

September 9, 2025

Open Universiteit
de beste! www.ou.nl

Abstract

This research investigates the possibilities of using Inductive Learning for Answer Set Programs (ILASP) and Clingo to learn the rules and strategies of the game of Connect-Four. Currently, there is not much knowledge about ILASP as a tool for learning logical programs. This limits its use, while it can have advantages over traditional machine learning techniques.

The game of Connect-Four is formulated using the Game Description Language and a reference program is created. Subsequently the rules were learned individually from selected examples generated by a custom made web app. These rules were merged into a program that was tested and validated. Finally some simple strategies were learned and added to the program.

The learning tasks show that ILASP is capable of learning rules and simple strategies similar to a reference program, and sometimes offers unexpected solutions. It is however challenging when the search space is large. Constraints in the GDL framework, which only examines one move at a time, limit the system's ability to learn complex strategies. Further documentation on ILASP could help lower the barrier to its adoption in learning logical programs and specifically game strategy learning.

Keywords: Inductive Logic Programming, Inductive Learning of Answer Set Programs, ILASP, Game Description Language, Connect-Four.

Source code from this research can be found on github:

https://github.com/boschloo/connect4

The custom web app used for generating game data is available on:

https://kladblok.app

CONTENTS

ACRONYMS

ASP Answer Set Programming. 5

clingo Potassco Solver program set. 5

GDL Game Description Language. 16

ILASP Inductive Learning of Answer Set Programs. 5, 6, 11

ILP Inductive Logic Programming. 5, 10

LP Logic Programming. 5

NAF Negation-As-Failure. 7

1. INTRODUCTION

Logic Programming (LP) is a successful declarative programming method for solving complex problems by describing solution properties using facts, rules, and constraints [Lif19]. Answer Set Programming (ASP), as a specific form of LP, has been applied successfully to a wide range of subjects, from AI, biomedical sciences, scheduling and planning to natural language processing. [EGL16]. A popular application area specially for programmers is game theory. LP enables the development of programs that can play games with clearly defined rules by finding all possible solutions.

For many types of problems, the rules are not immediately obvious. Inductive Logic Programming (ILP) was developed for this purpose. Given a certain background knowledge, a selection of facts (examples), and a declarative bias to restrict the hypothesis space, rules can be discovered to construct a logic program [MDR94]. ILP is used in the same areas of application as LP. One of the advantages of using ILP is the easy readability for humans and the explainability of the results [CD22].

This assignment was introduced and supervised by Jesse Heyninck as a graduation project for the Bachelor of Computer Science. It aligns with his research on practical applications of logical programming. The project started in collaboration with Myrthe Streep. The design and method of the research was developed jointly. Due to uneven progress, Myrthe already finished her part on the game of Go and graduated successfully [Str24].

The goal of my part of the research is to use ILP to learn the rules and possibly strategies of Connect-Four from game data only. Connect-Four is a well-known strategic board game in which two players using yellow and red discs take turns dropping a disc of their color into one of the columns on a vertical board. The objective is to be the first player to align four discs in a row, horizontally, vertically, or diagonally.

This study uses ASP as the LP language with the Potassco Solver program set (clingo) as its solver [GKK+11], and Inductive Learning of Answer Set Programs (ILASP) as a ILP system for machine learning [LRB15]. The research focuses on the techniques in learning with ILASP the rules of an arbitrary game. It is meant to provide insights in the practical usability of ILASP for learning and generating useful and efficient ASP programs.

Upon completion of the research, an ASP file will be learned with ILASP that enables a computer to play Connect-Four without prior knowledge of the rules. Also the question will be answered if it is possible to learn strategies that give a player a better chance of winning a game.

2. CONTEXT

This chapter provides background on the key concepts employed in this study. The terms ASP and ILASP are explained using examples, and elements of Game Description Language are mentioned as far as they are used in this research. The game of Connect-Four is introduced, and a global overview is given of relevant research in the field of learning games using Inductive Learning of Answer Set Programs (ILASP). Finally, the research questions are formulated and motivated.

2.1. ANSWER SET PROGRAMMING (ASP)

Declarative programming emphasizes specifying what a valid solution is, rather than how to compute it. In logic programming, a problem domain is described by facts and rules that derive new information. Answer Set Programming (ASP) extends this paradigm by computing sets of answers, each of which yields a consistent solution that satisfies all constraints. Answer Set Programming was developed around the beginning of this century [Bar03]. An ASP system such as clingo first grounds the program by instantiating variables to produce a propositional logic representation and then solves it by identifying stable models. Clingo, from the Potassco collection, integrates both grounding and solving to efficiently generate answer sets [GKKS22].

Given the right set of rules, facts and constraints, clingo yields all possible solutions for the problem at hand. Unlike imperative programming, there is no need to describe an algorithm to find a solution. This approach can be a bit abstract, but often it is more clear and easier to understand.

In the next paragraphs, some background is given on the concepts and the language used with ASP/clingo. The example in paragraph 2.1.4 shows the use of ASP/clingo in a simple example. A second example in paragraph 2.2.2 is about ILASP learning the rules of the first example. ILASP is the ILP-tool we used for learning rules and possibly strategies starting with game play situations.

2.1.1. SYNTAX OF ASP

- In ASP, a logic program is a finite set of rules constructed over a set of atoms. An **atom** is the most basic element of a logic program, representing a proposition or a fact within the problem domain. Atoms can take simple forms, such as propositional variables like p, which can be true or false. Typically, an atom is expressed as a predicate applied to a set of terms, for example, `parent(steve,lisa)` where `parent` is the predicate and `steve` and `lisa`, are terms.

- Atoms can include **constants** and **variables** as terms. For instance, the atom `parent(X,Y)` uses variables `X` and `Y`.
- Atoms and their negations are known as **literals**. Negations will be explained later on. With literals, ASP constructs rules, which are the fundamental syntactic units.
- A **rule** typically has a head and a body, both composed of literals. The symbol :-, the rule operator, links head and body and defines the rule. It is pronounced 'if'. The rule `daughter(Y,X):-parent(X,Y),female(Y).` can be read as "if `X` is the parent of `Y` and `Y` is female, then `Y` is the daughter of `X`" The head of the rule `daughter(Y,X)` is what is concluded if the body `parent(X,Y), female(Y)` is true. In an ASP program, all rules end with a period '.'.
- So a literal can either be an atom or its negation. This negation can be expressed in two forms: Negation-As-Failure (NAF) and classical negation.
  - Negation-As-Failure succeeds when there is no way to derive the atom, effectively treating it as false by default. For example: `trusted(X) :- person(X), not thief(X).` means "if X is a person and we cannot prove thief(X), then assume X is trusted."
  - Classical negation (`-`) explicitly declares an atom false. For example: `-trusted(X) :- thief(X).` reads "if X is a thief, then X definitely cannot be trusted."

  Both forms are prefix-operators. In practice, Negation-As-Failure is used more frequently in ASP models, since many problems rely on default assumptions rather than explicit refutations.
- A **program** is a set of rules. During the grounding phase of ASP, variables are replaced with all possible constants from the program's domain to generate a set of ground atoms. This process creates specific instances of general rules, which the solver then evaluates.

2.1.2. SEMANTICS OF ASP

The semantics of ASP revolves around how these rules are understood by the ASP solver to produce answer sets, which are the solutions to the given problem. The solver interprets rules in terms of logical implications: if the body of a rule is true, then the head must also be true. This logical framework allows the solver to systematically explore possible combinations of facts and rules to identify consistent sets of literals, known as answer sets. ASP rules can be categorized into three types: facts, normal rules, and constraints, each with its semantic role.

- **Facts** are rules with just a head and no body, representing known, unconditional truths. For instance, `father(steve).` and `parent(steve,lisa).` are facts that the solver assumes to be true without further justification.
- **Normal rules** define relationships between facts, specifying conditions under which new facts can be derived. For example, the rule `daughter(Y,X) :- parent(X,Y), female(Y).` states that a fact about a daughter can be inferred if the conditions in the body hold true.
- **Constraints** are rules with an empty head, which act as filters to eliminate undesirable answer sets. For instance, the constraint `:- father(bill).` ensures that any answer set containing `father(bill)` is invalid, effectively excluding it from the solution space.

2.1.3. Additional Constructs

ASP also includes choice rules, constants, weak constraints and directives that influence both syntax and semantics, adding flexibility and expressiveness to the language.

- **Choice rules** allow the programmer to specify that a certain number of literals from a given set must be true. For example, the rule `1 {p(1);p(2);p(3)} 2.` describes subsets of `{p(1),p(2),p(3)}` containing one or two elements, producing several (6) possible answer sets. This rule can also be written more concisely as `1{p(1..3)}2.` The solver interprets these choice rules to generate all valid combinations that satisfy the constraints, making choice rules a powerful tool for encoding combinatorial problems.
- **Constants** serve as placeholders for specific values, which can be defined within the program or provided as runtime input. For example, the directive `#const n=8.` sets a constant value n to 8, which can be used throughout the program, such as to define the dimensions of a chessboard. Constants can also be passed as command-line arguments when running the ASP solver, allowing for dynamic adjustments without altering the program's core logic.
- **(Hard) constraints** As mentioned before, a hard constraint is a rule of the form `:- Body.` An answer set in which `Body` holds is rejected.
- **Weak constraints** A weak constraint is an ASP rule of the form

```
:~ Body.[Weight@Level]
```

that may be violated but imposes a penalty (of given weight and priority) on a valid answer set. Answer sets are then ranked to minimize the total penalty. Weak constraints can be used to find an optimal solution when there is a preference for certain properties.

- Finally, the **directive** `#show` instructs the solver on which predicates to include in the output. `#show parent/2` for example, where 2 represents the arity of the predicate. This control over the output allows for a more focused and relevant presentation of the answer sets.

2.1.4. ASP: AN EXAMPLE

The following example is intended to demonstrate how to code a problem with ASP/-clingo. Suppose you want to serve fruit for lunch for a group of people. On the table are apples, pears, bananas,... and donuts? But the donuts appear to be gone before the start of the meal. Your guests may want one or two items. What combinations can be made when donuts are no longer available? This problem is solved by describing it in an ASP language using clingo as a solver.

First the items are summed up as facts:

```
item(apple). item(pear). item(banana). item(donut).
```

The choice can be expressed in a choice rule:

```
1 { take(V1) : item(V1) } 2.
```

To filter out the answer sets that do not meet the stated conditions, we use constraints. No more donuts available, so all answer sets containing a donut will be skipped:

```
:- take(donut).
```

Any fruit not taken is to be left on the table: a 'normal' rule:

```
left(X) :- not take(X), item(X).
```

To print the results we want, we add directives for the solver:

```
#show take/1.
#show left/1.
```

The complete program is saved as lunch.lp:

```
1 item(apple; pear; banana; donut). % syntactic sugar added
2 1 {take(V1) : item(V1)} 2.
3 :- take(donut).
4 left(X) :- not take(X), item(X).
5 #show take/1.
```

```
#show left/1.
```

Listing 1: lunch.lp - ASP demo program.

Running this program with clingo generates all possible answer sets, with items in random order.

```
clingo version 5.4.0
Reading from /Users/boschloo/c4/lunch.lp
Solving...
Answer: 1
left(donut) left(apple) {take(pear) left(banana)
Answer: 2
left(donut) left(apple) left(pear) take(banana)
Answer: 3
left(donut) left(apple) take(pear) take(banana)
Answer: 4
left(donut) take(apple) left(pear) left(banana)
Answer: 5
left(donut) take(apple) left(pear) take(banana)
Answer: 6
left(donut) take(apple) take(pear) left(banana)
SATISFIABLE

Models       : 6
Calls        : 1
Time         : 0.002s (Solving: 0.00s 1st Model: 0.00s Unsat: 0.00s)
CPU Time     : 0.001s
```

Listing 2: lunch.lp - ASP demo program, output.

So there are six possible combinations of one or two fruits from a collection of three fruits available.

2.2. INDUCTIVE LOGICAL PROGRAMMING

In the concept of Inductive Logic Programming (ILP), the term inductive is related to inductive reasoning: drawing general conclusions from specific examples. ILP is a subfield of machine learning that focuses on learning logical definitions from observed data. An ILP algorithm learns from positive and negative examples of a target concept, usually represented in Prolog or a similar language. It uses background knowledge consisting of facts and rules to provide context. The goal is to generate a hypothesis in the form of a logical program that explains all the positive examples without covering any of the negative ones. The search space of possible hypotheses can be defined and narrowed by using a language bias, which imposes restrictions on the form of the hypotheses.

2.2.1. ILP WITH ILASP

The system we used is called Inductive Learning of Answer Set Programs (ILASP) [LRB15]. ILASP is a framework developed largely by Mark Law from the Imperial College London. It follows the general design of ILP: given a set of examples, find a logic program that models them. Features of ILASP include its ability to generalize from very few examples, possibly using existing knowledge. The learned knowledge can easily be translated into English.

An ILASP program consists of three parts:

- Background knowledge. In general, an ILASP program can contain most types of rules from an ASP program.
- Hypothesis-space declarations. Directives defining the search space of rules ILASP may find. The hypothesis space can be explicitly defined, in the form length ~: rule, or by means of a mode declaration (mode bias).
  - #modeh declares allowed heads
  - #modeb declares allowed body literals.
  - #constant enumerates which constants (text, numbers) may fill const(..) placeholders
  - #maxv, #maxpenalty are directives to restrict sizes or numbers.
- Example declarations. ILASP learns from examples, which specify atoms that should or shouldn't appear in answer sets, sometimes within a specific context. Examples are either #pos (positive) or #neg (negative). Examples are in the format #pos(id,{e_1},{e_2},{context}), but not all parameters need to be present.
  - A positive example `#pos(e1,e2)` is satisfied if some answer set of $B \cup H$ includes all atoms in e_1 and excludes those in e_2.
  - A negative example `#neg(e1,e2)` is satisfied if no answer set of $B \cup H$ includes all atoms in e_1. Negative examples usually result in a constraint to be added.
  - An example can be context-dependent. The context is given as the last parameter.
  - To define preferences, there can be ordering examples as well. For this, the id has to be present in the examples.

When working with ILASP, tuning will be necessary to restrict run times. Parameters can be set in different locations. These will be explained where relevant.

2.2.2. ILASP: AN EXAMPLE

Here is an example how to learn the lunch problem from par. 2.1.4 using Inductive logic Programming (ILP). The goal is again, to show all possible combinations of one and two fruits for a lunch package. With ILP we start with a background and we choose some examples to deduce the rules. ILASP is used for ILP and clingo for testing and validating results.

First, the background is defined, in this case the items available for the examples:

```
item(apple). item(pear). item(donut).
```

Now we choose a number of examples, characteristic situations to be described (#pos or #neg).:

1.You can take just an apple, no pear:

```
#pos({take(apple)}, {take(pear)})
```

2.A combination of two is allowed, apple and pear for instance.

```
#pos({take(apple),take(pear) }, {}).
```

3.Donuts have disappeared long before lunch:

```
#neg({take(donut)}, {}).
```

4.At least one item should be taken:

```
#neg({},{take(apple),take(pear),take(donut)}).
```

Then the search space is defined. Because there are choices to be made, ingredients for a choice rule are chosen:

```
#modeha(take(var(v))).
#modec(item(var(v))).
#modeb(1, item(var(v))).
#modeb(1, take(donut)).
#maxv(1).
```

maxv(1) specifies that just one variable may be used in each rule. This restricts the size of the search space and the necessary runtime. The search space is generate by ILASP based on the given mode bias, and consist of 54 rules:

```
0  (base) ~/c4 % ilasp --version=2i lunch_las.lp -s
1  1 ~  :- take(donut).
2  1 ~  :- not take(donut).
3  1 ~  :- item(V1).
4  2 ~  :- item(V1); not take(donut).
5  2 ~   1 {take(V1) : item(V1) } 1.

50 .....
```

```
6 ~ 0 {take(V1) } 2 :- item(V1); not take(donut).
6 ~ 0 {take(V1) : item(V1) } 2 :- take(donut).
7 ~ 0 {take(V1) : not item(V1), item(V1) } 2 :- take(donut).
7 ~ 0 {take(V1) : not item(V1), item(V1) } 2 :- not take(donut).
```

Listing 3: lunch_las.lp - ILASP demo program, search space.

The complete program is saved as lunch_las.lp:

```
% Background
item(apple). item(pear). item(donut).

% Mode bias
#modeha(take(var(v))).
#modec(item(var(v))).
#modeb(1, item(var(v))).
#modeb(1, take(donut)).
#maxv(1).

% Examples
#pos({take(apple)}, {take(pear)}).
#pos({take(apple),take(pear) }, {}).
#neg({take(donut)}, {}).
#neg({},{take(apple),take(pear),take(donut)}).
```

Listing 4: lunch_las.lp - ILASP demo program.

Running this program gives the following output:

```
(base) ~/c4 % ilasp --version=2i lunch_las.lp
:- take(donut).
1 {take(V1) : item(V1) } 2.

(base) ~/c4 % ilasp --version=4 lunch_las.lp
1 {take(V1) : item(V1) } 2 :- not take(donut).
```

Listing 5: lunch_las.lp - ILASP demo program, output.

ILASP always uses V1, V2 etc. as variable names. The learned program with ILASP version 2i (listing 5, lines 2,3) is essentially the same as the ASP-example (listing 1, lines 2,3). With ILASP version 4 the two rules are combined. This shows that ILASP can learn regularities in just a few examples. However, care must be taken that all possible scenarios are present. Validation of results on datasets for testing is always necessary.

2.3. CONNECT-FOUR

Connect-Four is a well-known game made commercially by Milton Bradley (1974), later on Hasbro. It is played by two players on an upright board of seven columns and six rows. Each turn, a player drops a coin-size disk into one of the columns. Players use different colors, red and yellow. Due to the position of the board, the disk will end at the lowest possible position. For both players the goal is to get four disks of the player's color in one line, horizontal, vertical or diagonal. So each player has a different end state, with concrete properties. For each end state a large number of paths are possible in the form of sequences of moves. The choices that are to be made can be the result of a specific strategy. The game of Connect-Four is 'solved': at the start of the game, the outcome is already determined, assuming both players play optimally [All88], [All10].

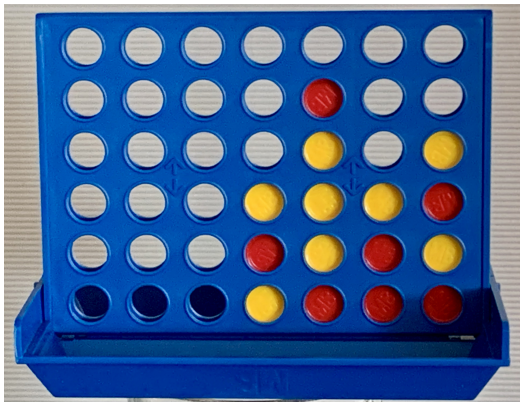

Figure 1: MB Spiele Connect-Four 'Vier gewinnt'

2.4. RELATED WORK

Connect-Four was mathematically solved by Victor Allis, and published in his doctoral thesis [All88], but James Dow Allen had found a solution just weeks before. Allen did not publish apparently until much later [All10]. Recently, a strong solution of Connect-Four with a look-up table was published by [Bö25].

General game playing is a framework for describing games and playing games by computers. It is presented in [GLP05], and formally defined in [LHH+08].

A standard work on Inductive Logic Programming is [MDR94], with an up-to-date version in [CD22]. [Lif19] gives a practical guide to Answer Set Programming. A more theoretical approach can be found in [GKKS22]. In [Law18] Mark Law presents the ILASP system he developed. Much information can be found on his website [LRB15].

In the current literature, [Gre18] uses ILASP for learning strategies through weak constraints. He focuses on Onamita, a strategy game, and reaches the conclusion that ILASP can learn strategies that are good in competition, but only from examples created with help of a human player. [SH22] uses also game theory. She finds ILASP having difficulties in finding a conclusive strategy for TicTacToe. [Dre23] explores the possibilities of ILASP by modeling multiple puzzles and investigating how different datasets influence the learning process. With ILASP it turns out to be difficult to learn the rules properly with only general examples. Specific guidance was needed to achieve a desired outcome.

2.5. RESEARCH QUESTIONS

In most research papers, not just ILASP but also other techniques are being used in finding solutions. This research, set up in collaboration with Myrthe Streep, extends the current literature by attempting to learn the game of Connect-Four using only standard ILASP-versions and using mode declarations to define predicates and terms of the mode bias. The main research question of this study is therefore:

RQ To what extent is it possible to learn Connect-Four using ILASP and clingo?

ILP research is scientifically interesting because it learns in a way that resembles machine learning methods, but using logical predicates. [MDR94]. Understanding its possibilities and constraints helps to determine for which applications it could be useful. Currently there is limited literature on ILASP, restricting knowledge of its capabilities and constraints. This makes it challenging to understand ILASP's potential when learning a game from game data. However, it is known that the reverse process of generating game sequences from given rules is possible using ASP. To explore the capabilities of ILASP, this research aims to learn an ASP program for the game of Connect-Four.

In our case, we are interested in whether the rules of a game can be learned by ILASP from game situations only. Therefore it is necessary to collect game play data, manually or automatically. The resulting ASP program should be tested in a formal and experimental way. This led us to the following sub questions:

SRQ1 How can we generate a selection of useful game data to use in ILASP?

SRQ2 To what extent is it possible to learn rules for Connect-Four from game data using ILASP and clingo?

SRQ3 To what extent is it possible to learn strategies for Connect-Four from game data using ILASP and clingo?

3. METHODOLOGY

Our research takes an exploratory approach to examine how well game rules and strategies can be learned from game data using ILASP and clingo. This chapter explains the methodology used. Section 3.1 covers how the game rules are formalized, while Section 3.2 describes how game data will be generated. Section 3.3 focuses on strategies. Section 3.4 outlines the methods to learn the game, and finally Section 3.5 explains how the learned rules and strategies will be validated.

3.1. FORMALIZING THE GAME RULES

The first thing that needs to be considered is how to model the game. Game Description Language (GDL) is a method that is often used to describe a game so a computer can play it without human intervention. GDL uses a set of predicates with a defined relation to each other. The predicates are the same in all game playing situations, but can be extended for a specific game. The predicates according to [LHH+08] that are used in this paper:

- `role(P)` shows the players P that participate in the game.
- `true(F)` shows that fact F holds in the current state of the game.
- `init(F)` shows that fact F holds in the initial state of the game.
- `legal(P,A)` is the action A that player P can take in the current state.
- `does(P,A)` is the action A that player P takes.
- `next(F)` shows that fact F holds in the next state of the game.
- `goal(P,S)` shows at the end if player P won (S=100), lost (S=0) or drew (S=50).
- `terminal` holds when the game has ended.

In each time step of the game, one of the two players is in control, represented by `true(control(P))`. The current state of the board is defined by predicates like `true(cell(X,Y,P))`, which indicate the position and ownership of each cell. Based on this state, all legal actions can be determined. A choice rule is then used to select one of these actions, and the consequences of that action are computed. The resulting state is described using `next(cell(X,Y,P))` for updated cell positions and `next(control(P))` for the next player in control. Each possible action leads to a different outcome, resulting in a separate answer set — up to seven for a game like Connect-Four.

Our approach models the game at a specific moment, capturing only the result of a single move. To simulate a full game using GDL, a General Game Playing Player (GGP Player) would be needed to manage the progression of time. A GGP Player uses the `init` predicate to compute the set of facts true in the initial state of the game. The `true`

predicates are the base for making a move by a player. At each time step, as a move is chosen, by the player or by the system, the facts that are true in the next state of the game are defined. The GGP Player takes care of the flow from one timestep to the next. This process is repeated for each subsequent move [LHH+08].

Before starting the learning process with ILASP, an ASP program will be made as a reference, to internalize the rules as well as the GDL framework (see Appendix A). This approach will help us to understand how the rules can be learned, gaining insight into the literals and number of variables required.

3.2. GAME DATA

Game data is essential before starting the learning process, as examples are required for an ILASP program to learn the rules. Also, for testing purposes example games are needed. The game data we need mainly consists of valid board states. For testing purposes, it is sometimes necessary to provide a move, to be able to compare the calculated output to the expected output. The GDL framework focuses on individual game states. In contrast to the method used with Go, in this research on Connect-Four the original seven-by-six board size is used, as we suspected that the learned formulas would otherwise not be correct.

[All88] and [All10] are used as a resource and as an inspiration for game data. A Connect-Four web app[1] was made as a tool for playing the game, reproducing and saving documented games from literature, and visualizing game data. This javascript powered app is also used to simulate games using different strategies and producing input files with examples for ILASP to learn. A program description can be found in Appendix C, the program code is on github[2].

ILASP should be able to learn the rules with a limited number of examples. Based on our experiences in the preparation, we planned two different approaches. The first approach is using as few examples as possible by constructing examples that specifically show just the rule to be learned. The other approach is generating and using a large number of examples, filtering out only the most non-relevant ones. We will compare these approaches and hope to be able to draw conclusions.

[1] https://kladblok.app
[2] https://github.com/boschloo/connect4

3.3. STRATEGIES

For each game, strategic rules can be discovered. The purpose of such rules is, to win the game when the rules are followed. A strategy in a game like Connect-Four comes down to making a choice from the possibles moves. In order to make that choice, preferences are necessary. We will have to define our preferences and learn these as so-called 'weak constraints'.

In talking about strategies, a lot of terminology is used. For the strategies we want to implement, the next two terms are important:

Threat 'A threat for player A is a square which, if taken by player B, connects four of B's men' [All88, p.17].

Check 'A cell available for immediate play whose occupation would complete four-in-row for one of the players' [All10, p.3]. The term check is analogous with chess.

The webapp also can show threats: the button Red's Threats shows the threats for the Black player. Strictly speaking, some of them are checks; the app does not distinguish between these concepts.

The preferences we want to implement are the following:

1. If the player in control has a check cell, the move should be in the column of the check cell so the player wins the game ('Win')
2. If the opponent has a check cell, the player in control should use the column of the check cell and prevent the opponent from winning in the next move ('Block').
3. When there are no check cells, the center columns of the board should be preferred to the edges ('Center')
4. If none of these conditions are met, the player in control should make a random choice ('Random').

These preferences were also added to the reference program which helped to get an understanding of the mechanism.

3.4. THE LEARNING PROCESS

With the framework selected and the examples identified, the learning process can start. The GDL predicates are organized in a specific manner within the ILASP file;

`role/1` in the background, `true/1` and `does/2` in the examples, leaving `legal/2`, `next/1`, `goal/2` and `terminal/0` to be learned. Because of the number of terms necessary, the resulting size of the search space, but also the unfamiliarity with ILASP, the learning process of these rules involved significant experimenting. The instructions drawn up by Myrthe [Str24, pg.8] were largely followed:

1. Place all relevant facts and rules in the background, including the rule or rules that need to be learned.
2. Select examples in the format `#pos({pos},{neg},{part}).` Use examples as partial interpretations in `{part}`.
   (a) Put a partial interpretation into the GDL file and use the output as `#pos`.
   (b) For `#neg`, use all possible outputs after removing the real output of 2a.
3. Run the ILASP program to verify the examples, aiming for the output of the rules already learned (initially empty).
4. Remove the target rule(s) of this iteration from the initial background.
5. Adjust the mode bias to enable it to learn the rule(s).
   (a) Add rule heads with `#modeh`. Use placeholders for variables and constants.
   (b) Add bodies with `#modeb` where the first argument is the expected maximum number of occurrences in the rule. Use placeholders for variables and constants.
   (c) Set the maximum number of variables in a rule with `#maxv`.
   (d) If more than three literals are expected in the body, include the parameter `-ml=[integer]` on the command line. If there will be more than four, add `–max-rule-length=[integer+1]`.
6. Run ILASP and verify the target rule has been learned.
7. The command line parameter `-s` will show the hypothesis search space constructed with the given mode bias. If the rule expected to be learned is not there, there may be an error in the mode bias, or the maximum allowed number of variables and literals can be too low.

For learning the preferences, some specific other keywords were used for the mode bias, and an ordering of the examples has to be given.

3.5. VALIDATION

After learning the rules with ILASP, they can be merged and tested. This involves combining the learned rules with the background into one ASP file and adding the action selection rule. The test scenario's (fig.2) are based on the normal flow of a game, and focus on boundary conditions like winning moves.
First we test the initial background situation, and the concept of a line (four in a row). Then we can test the properties of the different scenario's on the board:

1. The empty board
2. The board with columns filled differently, including empty columns and completely filled columns
3. The board is filled, except for one cell
4. The board is filled completely

Three of these can be combined with a winning condition:

1. The player in control cannot make a winning line
2. The player in control can make a winning line
3. The player before has made a winning line.

It was interesting to test these combinations and see how they give the correct outcome for the predicates open, target, legal, does, next, line, goal and terminal. Testing the learned rules provided also insight into how the GDL works.

4. RESULTS AND DISCUSSION

The reference program is covered in section 4.1. The results of the learning process are given in section 4.2 for the game rules, in section 4.3 as concerning the strategies. Section 4.4 describes the validation.

4.1. RESULTS

In a similar manner to the learning of the game of Go [Str24], the learning of Connect-Four is discussed here. Because the learning of multiple rules at once was found to be time-consuming and error-prone, rules were learned individually. Attempts to merge these learning tasks later on were also not successful because of exponentially increasing run times. The given rules as well as the rules learned are discussed below. The final programs used to learn the rules are available on the github site.

Testing Connect-Four ASP

#	Background	board_size/2.	col/2.	row/1.	init/1.	init/3.	cell/3.	control/1					
1	Background facts, initial game state	X	X	X	X	X							
2	Start of the game						X	X					
	Lines	checks	line							line/1.			
3	Board with horizontal line in cell/3.	yes	yes							1			
4	Board with vertical line in cell/3.	yes	yes							1			
5	Board with diagonal line SW-NE in cell/3.	yes	yes							1			
6	Board with diagonal line NW-SE in cell/3	yes	yes							1			
7	Board with horizontal and diagonal line SW-NE	yes	yes							2			

	Game states	checks	line	control/1.	open/0.	open/2.	target/3.	terminal/0.	legal/2.	line/1.	next/1	next/3	goal/2.
8	Start of the game, all columns empty	no	no	1	T	7	7	F	7	F	2	X	50/50
9	Board one column full, six columns available	no	no	2	T	6	6	F	6	F	1	X	50/50
10	Board almost full, one column open	no	no	2	T	1	1	F	1	F	1	X	50/50
11	Board seven columns filled completely	no	no	1	F	0	0	T	0	F	2	X	50/50
12	Board one column full, six columns available	yes	no	1	T	6	6	F	6	F	2	6	50/50
13	Board almost full, one column open	yes	no	2	T	1	1	F	1	F	1	1	50/50
14	Board one column full, six columns available	no	yes	2	T	6	6	T	0	T	1	1	100/0
15	Board almost full, one column open	no	yes	2	T	1	1	T	0	T	1	1	100/0
16	Board seven columns filled completely	no	yes	1	F	0	0	T	0	T	2	1	0/100

Figure 2: Test scenarios

4.1.1. REFERENCE PROGRAM

First a reference program was constructed and tested. While working with the GDL-framework, for the sake of clarity the predicates were somewhat simplified. The predicate true(control(P)) is written as control(P), true(cell(X, Y, P)) as cell(X, Y, P), and next(cell(X, Y, P)) as next(X, Y, P). This also simplified the mode bias to be used in ILASP later on.

All predicates from the GDL framework have been worked out, plus a few helper predicates: target/2 to define cells that can be filled on the next move, open/1 to indicate a column available for a move, and opponent/2, to show the other player. The idea behind this was that helper predicates can simplify the learning process.

The input of the program is a game state, consisting of a game board status and a player in control. The game board is given as a matrix of cells: cell(X, Y, P). Using a standard 7x6 game board these are 42 cell facts. The board_size/2 is given as a fact, also the player roles as these are always the same for a standard game of Connect-Four. The corresponding ASP programs can be found in Appendix A.

Testing or this reference program was done manually. As the number of test routines was limited this worked well enough. We discovered that at the moment a player makes a move, it is not yet possible to find out if there is a winning situation (a line). Because when line/1 holds, also terminal/0 is true. And when a game is terminal, no legal move can be made. One has to wait until the next time step in the game to check for a line, or another terminal condition.

The web app (Appendix C) is used to provide input scenario's and to visualize the output generated by clingo.

4.1.2. BACKGROUND

The background of the ILASP-programs consists of facts and rules assumed to hold for all examples during learning. In all programs, `role(1)` and `role(2)` are given as background, also predicates that are used in all examples. Game states consist of 42 cell/3 predicates and are usually added as context-dependent instances.

4.1.3. RULES LEARNED

The rules learned will be discussed one by one, in order of their dependencies as shown in figure 3. With each predicate, learned rules in red are compared to the rules in the Reference program in blue text.The corresponding ILASP programs are on github.

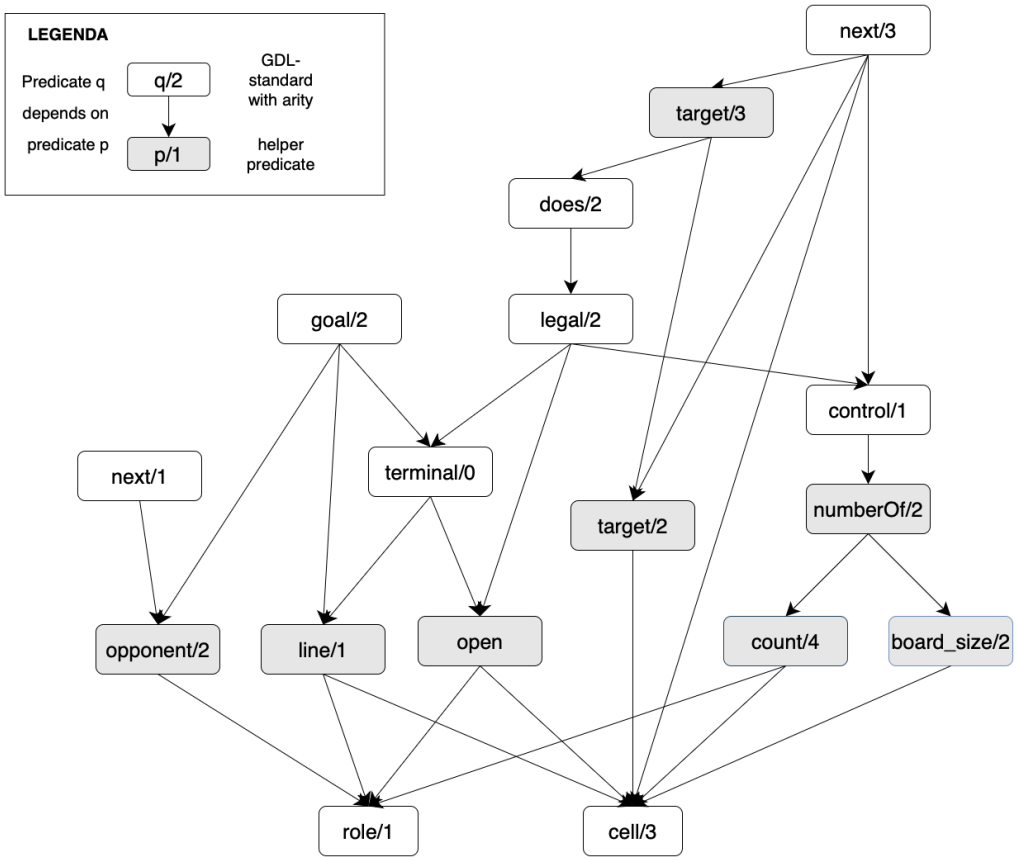

Figure 3: Dependency of the predicates/ rules learned.

Opponent/2

A predicate that may come in handy is opponent/2. Since there are two players, for each player the other one is the opponent. This can be learned by ILASP from this tiny code fragment:

```
role(1). role(2).

#modeh(opponent(var(r),var(r))).
#modeb(2, role(var(r))).
#modeb(1, var(r) < var(r)).
#maxv(2).

#pos({opponent(1,2), opponent(2,1)},{opponent(1,1)},{}).
```

ILASP gives the following rules for opponent/2:

```
opponent(V1,V2) :- role(V1); role(V2); V1 < V2.
opponent(V1,V2) :- role(V1); role(V2); V2 < V1.
```

However, when we change the mode bias and add var(r)!=var(r), just one line is returned. Summarized:

```
Reference:
    opponent(1,2). opponent(2,1).
Learned:
    opponent(V1,V2) :- role(V1); role(V2); V1 < V2.
    opponent(V1,V2) :- role(V1); role(V2); V2 < V1.
or:
    opponent(V1,V2) :- role(V1); role(V2); V2 != V1.
```

Listing 6: Opponent as learned with ILASP compared to the reference program.

ILASP learns rules, while the Reference program uses facts. It is not possible to learn facts with ILASP.

Open/0 and open/1

In order to properly learn the `terminal` rule, a rule for open was deemed useful. A column is called open if it can accommodate at least one additional disk. Two versions of open are learned: `open/1` and `open/0`. The last one can indicate the end of the game, when 42 moves have been made.

```
Reference:
    open(X) :- cell(X,N,0), board_size(M,N).
    open :- open(X).
Learned:
    open(V1) :- cell(V1,V2,0).
    open :- cell(V1,V2,0).
```

Listing 7: 'Open' as learned with ILASP compared to the reference program.

Line/1

A game of Connect-Four ends when one of the players has four discs in a row, this is called a line. A line can be in one of four directions.

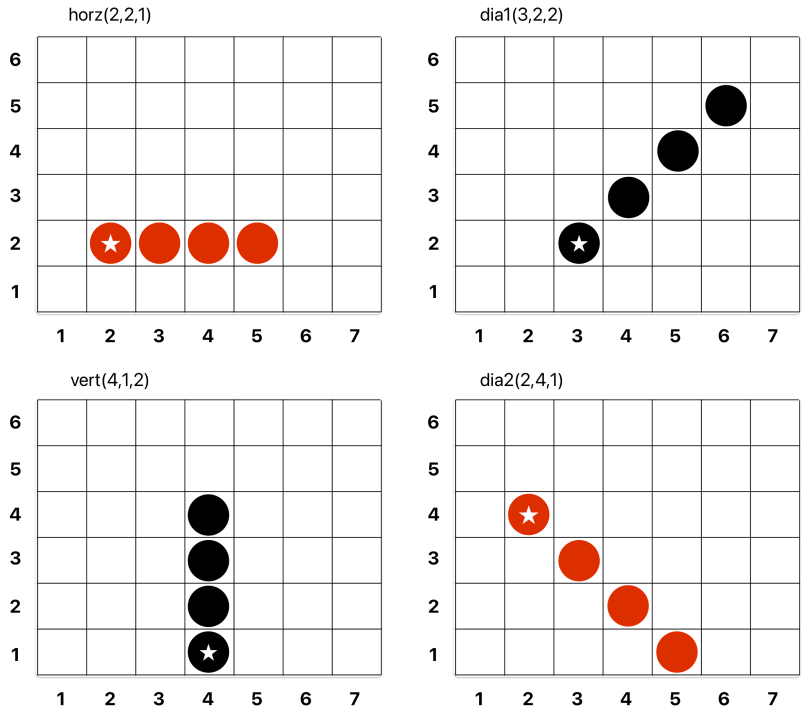

Figure 4: Lines, directions and starting points

Modeling the lines took a lot of effort. There are several ways to model the game board with the lines, but no matter how, a lot of variables are involved. In the end, we chose to use four cells with X and Y coordinates and a player role. 'Real' games were used as examples, simulated or played by hand. There has been a selection of the game boards so the complete rules could be learned with a limited number of examples. The rules learned this way are:

```
Reference:
     horz(X,Y,P) :-cell(X,Y,P), cell(X+1,Y,P), cell(X+2,Y,P), cell(X+3,Y,P), role(P).
     line(P) :- horz(X,Y,P).
Learned:
     horz(V1,V2,V3) :- role(V3); cell(V1,V2,V3); cell(V1+1,V2,V3); cell(V1+2,V2,V3);
         cell(V1+3,V2,V3).
     line(V1) :- horz(V2,V3,V1).
```

Listing 8: Lines as learned with ILASP compared to the reference program.

Predicates vert/3, dia1/3 and dia2/3 are learned accordingly and can be found in the github archive.

For horz/3 and vert/3, one example was sufficient, for dia1/3 and dia2/3 two examples were necessary to get a rule that tested well. The learned rules are substantively the same as the ones in the reference program.

Board_size/2

An attempt was made to learn the rules to find the size of the board given a correct game board state. In ASP, the following code fragment would do the job:

```
cols(C) :- C = #max{X :cell(X,_,_) }.
rows(R) :- R = #max{Y :cell(_,Y,_) }.
board_size(C,R) :- cols(C), rows(R).
```

Listing 9: board_size/2. constructed as reference.

However, it is not possible to learn aggregate functions with ILASP.

A number of game examples and a mode bias for board_size/2 were constructed with a background as small as possible. Through experimentation the number of examples was reduced and the mode bias extended with some features. The resulting ILASP program is shown below:

```
% c4_v705_size_las.lp
% Return the board size.

role(1).role(2).
col(X) :- cell(X,_,_).
row(Y) :- cell(_,Y,_).

#pos({size(7,6)},{
size(1,1), size(2,1), size(3,1), size(4,1), size(5,1), size(6,1), size(7,1),
size(1,2), size(2,2), size(3,2), size(4,2), size(5,2), size(6,2), size(7,2),
size(1,3), size(2,3), size(3,3), size(4,3), size(5,3), size(6,3), size(7,3),
size(1,4), size(2,4), size(3,4), size(4,4), size(5,4), size(6,4), size(7,4),
size(1,5), size(2,5), size(3,5), size(4,5), size(5,5), size(6,5), size(7,5),
size(1,6), size(2,6), size(3,6), size(4,6), size(5,6), size(6,6),
size(1,7), size(2,7), size(3,7), size(4,7), size(5,7), size(6,7), size(7,7)
},{
cell(1,6,0).cell(2,6,0).cell(3,6,0).cell(4,6,0).cell(5,6,0).cell(6,6,0).cell(7,6,0).
cell(1,5,0).cell(2,5,0).cell(3,5,0).cell(4,5,0).cell(5,5,0).cell(6,5,0).cell(7,5,0).
cell(1,4,0).cell(2,4,0).cell(3,4,0).cell(4,4,0).cell(5,4,0).cell(6,4,0).cell(7,4,0).
cell(1,3,0).cell(2,3,0).cell(3,3,0).cell(4,3,0).cell(5,3,0).cell(6,3,0).cell(7,3,0).
cell(1,2,0).cell(2,2,0).cell(3,2,0).cell(4,2,0).cell(5,2,0).cell(6,2,0).cell(7,2,0).
cell(1,1,1).cell(2,1,0).cell(3,1,0).cell(4,1,0).cell(5,1,0).cell(6,1,0).cell(7,1,0).
}).

#modeh(size(var(x),var(y))).
#modeh(greaterX(var(x))).
#modeh(greaterY(var(y))).
#modeb(1, role(var(r))).
#modeb(1, greaterX(var(x))).
```

```
#modeb(1, greaterY(var(y))).
#modeb(2, col(var(x))).
#modeb(2, row(var(y))).
#modeb(1, var(x) > var(x)).
#modeb(1, var(y) > var(y)).
#maxv(2).
```

Listing 10: ILASP-program to learn board_size/2.

With this program the following rules were learned:

```
greaterY(V1) :- row(V1); row(V2); V2 > V1.
greaterX(V1) :- col(V1); col(V2); V2 > V1.
size(V1,V2) :- col(V1); row(V2); not greaterX(V1); not greaterY(V2).
```

Listing 11: board_size/2 as learned with ILASP.

Remarkable is the use of the predicates `greaterX/1` and `greaterY/1`. The origin of these predicates were rules given in the background of the ILASP program, but after removing them from the background and inserting plausible lines in the mode bias, ILASP uses them in the learned output.

Terminal/0

The learned terminal/0 rules are the same as the reference rules:

```
Reference:
    terminal :- line(P).
    terminal :- not open.
Learned:
    terminal :- line(V1).
    terminal :- not open.
```

Listing 12: Terminal/0 as learned with ILASP compared to the reference program.

Target/2, target/3

With each move, the player in control has at most seven possible moves to chose from: seven columns, as long as there is at least one empty cell in the column. To calculate the next cell contents for the board, it is important to know which cells can be filled. Because of gravity, a disc always ends up on the lowest empty cell in a column. In the program, these cells will be called targets. The rules for target/2 and target/3 are learned from examples. Target/2 consists of the coordinates of the lowest available cell. Target/3 also includes the role of the player in control.

```
Reference:
    target(X,Y) :- cell(X,Y,0), not cell(X,Y-1,0).
    target(X,Y,P) :- does(P, drop(X)), cell(X,Y,0), not cell(X,Y-1,0).
Learned:
    target(V1,V2) :- cell(V1,V2,0); not cell(V1,V2-1,0).
    target(V1,V2,V3) :- target(V1,V2); does(V3,drop(V1)).
```

Listing 13: Target/2 and target/3 as learned with ILASP compared to the reference program.

Goal/2

`Goal/2` also is learned with a few simple examples. Results are varying, depending on which helper predicates are used. The most elegant rules were learned using `terminal/0` in the examples and in the bias:

Listing 14: Goal/2 as learned with ILASP compared to the reference programcaptionpos

```
Reference:
    goal(P,100) :- line(P).
    goal(P, 50) :- not goal(P,100), not goal(P,0), role(P).
    goal(P,  0) :- goal(P2,100), P!=P2, role(P).
Learned:
    goal(V1,100) :- line(V1).
    goal(V1,50) :- opponent(V1,V2); not terminal.
    goal(V1,0) :- line(V2); opponent(V1,V2).
```

Legal/2

The predicate `legal/2` defines the actions players are allowed to. On each time step, one player is in control. The other player has no legal actions. It is possible to model this as a legal action 'noop' but for Connect-Four this does not offer any benefits. With the right mode bias, the rule learned is identical to the reference rule.

Listing 15: Legal/2 as learned with ILASP compared to the reference program.captionpos

```
Reference:
    legal(P, drop(X)) :- control(P), open(X), not terminal.
Learned:
    legal(V1,drop(V2)) :- control(V1); open(V2); not terminal.
```

Control/1

When the game starts, player 1 is the first to make a move. So when there are no discs, player 1 is in control. Because exactly one disc is added with each move, the number of discs on the board determines whose turn it is. To learn this with ILASP from a number of examples, a counting function could be useful. ILASP however cannot learn the aggregate functions that are available in clingo. Therefore, a predicate numberOf/2 is added to the background. It uses the cells/3 input to calculate the number of discs for the players by means of recursion:

```
% Predicate numberOf/2 ----------------------------
count(1,0,P,0) :- role(P).
count(X,Y,P,N) :- count(X,Y-1,P,N-1), cell(X,Y,P).
count(X,1,P,N) :- count(X-1,Y-1,P,N-1), not cell(X-1,Y,_), cell(X,1,_), cell(X,1,P).
count(X,Y,P,N) :- count(X,Y-1,P,N), cell(X,Y,_), not cell(X,Y,P).
count(X,1,P,N) :- count(X-1,Y-1,P,N), not cell(X-1,Y,_), cell(X,1,_), not cell(X,1,P).
numberOf(P,N) :- count(X,Y,P,N), board_size(X,Y).
```

Listing 16: ASP-program for numberOf/2.

Using this function and three examples, the rules for control/1 are learned:

```
Reference:
    % In the reference program, control/1 is given as a fact.
Learned:
    control(V1) :- numberOf(V1,V3); numberOf(V2,V3); V1 < V2.
    control(V1) :- numberOf(V1,V2); numberOf(V4,V3); V2 < V3.
```

Listing 17: Control/1 as learned with ILASP compared to the reference program.

Does/2

Does/2 is the base rule of the game. For the player in control, there are options to choose from. The corresponding ASP-rule should therefore be a choice rule. Does/2 depends on legal/2, and indirectly on various other rules that are given as background in the ILASP program. The rule learned is:

```
Reference:
    {does(X, A): legal(X, A)} = 1 :- not terminal.
Learned:
    1 {does(V2,V1) : legal(V2,V1) } 1.
```

Listing 18: Does/2 as learned with ILASP compared to the reference program.

Compared with the reference program, the 'not terminal' term is missing because it is already covered in the definition of legal/2.

Next/1

Next/1 indicates the player in control after a player's move. In the reference program, next/1 is given as facts; with ILASP a rule is learned based on opponent/2.

```
Reference:
    next(control(2)) :- control(1).
    next(control(1)) :- control(2).
Learned:
    next(control(V1)) :- opponent(V2,V1); not next(control(V2)).
```

Listing 19: Next/1 as learned with ILASP compared to the reference program.

Next/3

To learn next/3, the positions of the discs after the player's move, two approaches have been tried. The first ILASP program is set up without helpers. The second one uses the additional concepts target/2 and target/3. Both approaches gave good results with two extensive examples.

```
Reference:
    next(X,Y,P) :- target(X,Y,P).
    next(X,Y,P) :- cell(X,Y,P), role(P).
    next(X,Y,0) :- cell(X,Y,0), not target(X,Y,_).
Learned without helper predicates:
    next(V1,V2,V3) :- cell(V1,V2,V3); not cell(V1,V2,0).
```

```
    next(V1,V2,V3) :- cell(V1,V2,V3); cell(V1,V2-1,0).
    next(V1,V2,V3) :- cell(V1,V2,0); does(V3,drop(V1)); not cell(V1,V2-1,0).
    next(V1,V2,0) :- control(V3); cell(V1,V2,0); not does(V3,drop(V1)).
Learned using helper predicates:
    next(V1,V2,V3) :- target(V1,V2,V3).
    next(V1,V2,V3) :- role(V3); cell(V1,V2,V3).
    next(V1,V2,0) :- control(V3); cell(V1,V2,0); not target(V1,V2,V3).
```

Listing 20: Next/3 as learned with ILASP compared to the reference program.

The version with the use of helpers is almost the same as the reference version. For both solutions the meaning of the learned rules can be easily deduced. This is one of the aspects that makes Inductive Logic Programming transparent and explainable.

4.1.4. STRATEGIES LEARNED

To recapitulate briefly, the three strategies to learn are:

1. If in check, do the winning move ('Win')
2. If the opponent has check, block (('Block')
3. Choose one of the three center columns ('Center')

If none of the former conditions are valid, the program should choose one of the remaining columns ('Random'). To determine the checks and other threats, a new set of rules has been created manually and added to the reference program and to the background of the ILASP files.

```
% Vertical
check(P,X) :- cell(X,Y,P), cell(X,Y+1,P), cell(X,Y+2,P), cell(X,Y+3,0). % Just one option

% Horizontal
check(P,X) :- cell(X,Y,0),cell(X+1,Y,P),cell(X+2,Y,P),cell(X+3,Y,P),target(X,Y).
check(P,X) :- cell(X-1,Y,P),cell(X,Y,0),cell(X+1,Y,P),cell(X+2,Y,P),target(X,Y).
check(P,X) :- cell(X-2,Y,P),cell(X-1,Y,P),cell(X,Y,0),cell(X+1,Y,P),target(X,Y).
check(P,X) :- cell(X-3,Y,P),cell(X-2,Y,P),cell(X-1,Y,P),cell(X,Y,0),target(X,Y).

% diagonal SW-NE
check(P,X) :- cell(X,Y,0),cell(X+1,Y+1,P),cell(X+2,Y+2,P),cell(X+3,Y+3,P),target(X,Y).
check(P,X) :- cell(X-1,Y-1,P),cell(X,Y,0),cell(X+1,Y+1,P),cell(X+2,Y+2,P),target(X,Y).
check(P,X) :- cell(X-2,Y-2,P),cell(X-1,Y-1,P),cell(X,Y,0),cell(X+1,Y+1,P),target(X,Y).
check(P,X) :- cell(X-3,Y-3,P),cell(X-2,Y-2,P),cell(X-1,Y-1,P),cell(X,Y,0),target(X,Y).

% diagonal NW-SE
check(P,X) :- cell(X,Y,0),cell(X+1,Y-1,P),cell(X+2,Y-2,P),cell(X+3,Y-3,P),target(X,Y).
check(P,X) :- cell(X-1,Y+1,P),cell(X,Y,0),cell(X+1,Y-1,P),cell(X+2,Y-2,P),target(X,Y).
check(P,X) :- cell(X-2,Y+2,P),cell(X-1,Y+1,P),cell(X,Y,0),cell(X+1,Y-1,P),target(X,Y).
check(P,X) :- cell(X-3,Y+3,P),cell(X-2,Y+2,P),cell(X-1,Y+1,P) cell(X,Y,0),target(X,Y).
```

Listing 21: Checks /2, manually created

The following lines were also added to the reference program: a helper center/1 to define the center columns, and three weak constraints. The order of these weak con-

straints is to be determined by the 'penalty', this is the first number in brackets.

```
% Strategies: 1.Win 2.Block opponent 3.Choose center columns.
center(X) :- X > 2, X < 6, col(X).

:~ does(P,drop(X)), not check(P,X). [50@1]
:~ does(P,drop(X)), not check(O,X), opponent(P,O). [20@1]
:~ does(P,drop(X)), not center(X). [10@1]

#show does/2.
```

Listing 22: Strategies, manually created

The ILASP-part of the program:

```
% Game state ---------------------------------------------------------------
cell(1,6,0).cell(2,6,0).cell(3,6,0).cell(4,6,0).cell(5,6,0).cell(6,6,0).cell(7,6,0).
cell(1,5,0).cell(2,5,0).cell(3,5,0).cell(4,5,0).cell(5,5,0).cell(6,5,0).cell(7,5,0).
cell(1,4,0).cell(2,4,0).cell(3,4,2).cell(4,4,0).cell(5,4,0).cell(6,4,0).cell(7,4,0).
cell(1,3,0).cell(2,3,0).cell(3,3,2).cell(4,3,0).cell(5,3,0).cell(6,3,0).cell(7,3,0).
cell(1,2,0).cell(2,2,0).cell(3,2,2).cell(4,2,0).cell(5,2,0).cell(6,2,0).cell(7,2,0).
cell(1,1,1).cell(2,1,0).cell(3,1,1).cell(4,1,1).cell(5,1,0).cell(6,1,0).cell(7,1,0).
control(1).

% . . . . . . .
% . . . . . . .
% . . 2 . . . .
% . . 2 . . . .
% . . 2 . . . .
% 1 . 1 1 . . .

#modeo(1, does(var(r),drop(var(x))),(positive)).
#modeo(1, check(var(r),var(x))).
#modeo(1, opponent(var(r),var(r)),(anti_reflexive)).
#modeo(1, target(var(x),var(y))).
#modeo(1, role(var(r))).
#modeo(1, center(var(x))).

#maxp(3).
#maxv(3).
#weight(20).

#pos(p1,{},{},{control(1).does(1,drop(2)).}).
#pos(p2,{},{},{control(1).does(1,drop(1)).}).
#pos(p3,{},{},{control(1).does(1,drop(3)).}).
#pos(p4,{},{},{control(2).does(2,drop(3)).}).
#pos(p5,{},{},{control(2).does(2,drop(2)).}).
#pos(p6,{},{},{control(2).does(2,drop(6)).}).
#pos(p7,{},{},{control(2).does(2,drop(5)).}).

#brave_ordering(p1,p2,<).
#brave_ordering(p1,p3,<).
#brave_ordering(p4,p5,<).
#brave_ordering(p4,p5,<).
#brave_ordering(p5,p6,<).
#brave_ordering(p7,p6,<).
```

With this program the preferences were learned:

```
Reference:
    :~ does(P,drop(X)), not check(P,X). [50@1]
    :~ does(P,drop(X)), not check(O,X), opponent(P,O). [20@1]
    :~ does(P,drop(X)), not center(X). [10@1]
Learned:
    :~ does(V1,drop(V2)), not center(V2).[20@1, V1, V2]
    :~ does(V1,drop(V2)), not check(V1,V2).[20@2, V1, V2]
    :~ does(V1,drop(V2)), role(V3), not check(V3,V2).[20@3, V1, V2, V3]
```

Listing 23: Preferences learned with ILASP compared to the reference program.

4.1.5. VALIDATION

The rules learned with ILASP are combined with the necessary background predicates to a new ASP/clingo file. When there are multiple variations of the rules, the rule that is easiest to understand (which is subjective) is chosen. The complete learned program c4_asp_LEARNED.lp is shown in Appendix B.

Testing was performed according to the test scheme in fig.3. The testing process gradually became more exploratory, as errors were discovered and fixed.

Test results are to be found on github.

4.2. DISCUSSION

This study centered on ILASP learning the rules of the game of Connect-Four from a selection of examples. The learned rules were combined into one ASP/clingo file and verified through unit tests. The result of this research shows that ILASP is capable of learning the selected rules and strategies of the game of Connect-Four.

Relatively little attention was paid to collecting game data. Unlike the approach used in Go where a two-by-two board was used, in Connect-Four the examples are given for the entire 7x6 game board. A custom web app, kladblok.app with built-in simple strategies was used for this purpose. The examples were selected from randomly generated games, with the assumption that they should be representative of the rule to be found.

Learning individual rules was relatively easy and generally didn't take much time to run (less than 20 seconds). However, learning combinations of rules turned out to be time-consuming and prone to errors. In particular, learning the 'row-of-four' rule was challenging. To give an indication: the longest endured runtime was almost 24 hours, to learn 'row-of-four' in all four directions using 14 examples.[3]

[3]ILASP package version 4.4.0, program version 2i, clingo version 5.7.1, macBook Air M2, 16GB

The method used was to learn rules covering all variations with as few examples as possible. Sometimes this was done by learning with a larger number of examples and then eliminating those that provided no additional information. This procedure was followed as long as it resulted in a rule that proved to be valid. Sometimes just a few examples of game states were needed to create effective rules, sometimes many. It turned out that examples had to be chosen that specifically demonstrated the rule to be learned.

For constructing the bias mode there are just a few guidelines and examples. Also it was not clear beforehand which syntax ILASP permits in the ASP background.

Combining multiple examples and mode bias to learn multiple rules simultaneously, sometimes resulted in combinations from which the individual rules could no longer be unambiguously derived (returning UNSATISFIABLE). Using a larger number of examples might have been a solution, but due to time constraints, this approach was not successfully implemented. The size of the resulting search space and the longer run times were a concern also. In general, the system provides little feedback. As the number of variables increases, the required runtime can grow exponentially. ILASP does not show any indication on progress made.

For all experiments in learning ILASP v2i is used, as the extra options of later versions were not needed and the runtime was shorter in comparison. Myrthe Streep investigated this and reported on it in her part of our project [Str24].

During the process, various ways of modeling the game board and the lines were considered. Ultimately, using standard Cartesian coordinates for the board, along with separate rules for lines in each of the four directions, proved to be the simplest to learn and the easiest to understand.

Because the GDL standard for modeling the games was followed, it was not possible to learn more complex strategies. In GDL, only one move is examined at a time, while in strategies, the opponent's response plays an important role.

In the learning process it is essential to know what rules to learn and also how they should be learned. Without knowing which terms to use, it is nearly impossible to create a mode bias that will do the job. The downside of this, is that the rules we learned are often very similar to the reference rules. - Learning a few simple strategies however, based on known best-practices, turned out to be rather simple given the check conditions as background.

Working with ILASP was a mixed experience. Frameworks like ILASP are difficult to use without extensive knowledge of the underlying formal theory. Example cases could facilitate this, but for ILASP, such examples are hard to find.

When comparing Inductive Logic Programming to other Machine Learning methods, it is noticeable that with both techniques, patterns can be recognized in data and rules can be derived. Also predictions are possible based on training data. With Inductive Logic Programming, the results can be understood and explained in plain language, while, for example Neural Networks are more like a black box. In general ILP is able to learn rules with very few examples. Also, existing knowledge can easily be integrated as part of the background and used for further learning.

5. Conclusion and Future Work

This research aimed to answer the research questions: "To what extent is it possible to learn rules and strategies of the game of Connect-Four using ILASP and clingo?". To find the answer, exploratory research was conducted using the GDL framework to formulate the rules. Data was selected from game states generated by a web app using different playing strategies. The primary focus of the learning process was on iteratively learning with ILASP and resolving errors.

Using ILASP to learn the rules and strategies of the game of Connect-Four, the following conclusions can be drawn:

- With ILASP it is possible to learn rules similar to a given reference program. A number of times, the rules learned pointed out errors in the reference program. And incidentally it presented an unexpected solution.
- With ILASP it is possible to learn simple preferences when given representative examples.
- To obtain output within a reasonable time, it was necessary to tune the mode bias declarations with extensions or restrictions on the number of variables and the length of the rules.

Initially, the idea of following the GDL standard seemed like the easiest way to align with current research methods. However, when determining strategies, this proved to be a limitation, as it only considers a single move ahead. Other modeling techniques that incorporate time as a factor might be better able to utilize the potential of ASP/clingo, including for strategy learning.

A working version of the game with the learned rules and strategies using Clinguin or PyLASP, the ILASP Python extension, has not been completed due to time constraints.

Future research could be on exploring a different approach on strategies, using time steps as an extra dimension [SH22] to select winning paths.

When confronted with increasing run times, it appears that ILASP and clingo do not employ mechanisms for parallel computing. This could be a topic for future research, even though it cannot overcome the exponentially growing processing time.

Work could also be done to document the use of ILASP, so the threshold for use is lowered.

REFERENCES

[All88] Louis Victor Allis. A knowledge-based approach of connect-four. *J. Int. Comput. Games Assoc.*, 11(4):165, 1988. 14, 17, 18

[All10] James Dow Allen. *The Complete Book of Connect 4: History, Strategy, Puzzles*. Puzzle Wright Press, 2010. https://archive.org/details/isbn_9781402756214/ Accessed: July 2, 2024. 14, 17, 18, 46

[Bar03] Chitta Baral. *Knowledge representation, reasoning and declarative problem solving*. Cambridge university press, 2003. 6

[Bö25] Markus Böck. Strongly solving 7x6 connectfour on consumer grade hardware, 2025. https://arxiv.org/pdf/2507.05267 Accessed: July 18, 2025. 14

[CD22] Andrew Cropper and Sebastijan Dumančić. Inductive logic programming at 30: a new introduction. *Journal of Artificial Intelligence Research*, 74:765–850, 2022. 5, 14

[Dre23] Talissa Dreossi. Exploring ilasp through logic puzzles modelling. In *CEUR WORKSHOP PROCEEDINGS*, volume 3428. CEUR-WS, 2023. 15

[EGL16] Esra Erdem, Michael Gelfond, and Nicola Leone. Applications of answer set programming. *AI Magazine*, 37(3):53–68, 2016. 5

[GKK+11] Martin Gebser, Benjamin Kaufmann, Roland Kaminski, Max Ostrowski, Torsten Schaub, and Marius Schneider. Potassco: The potsdam answer set solving collection. *Ai Communications*, 24(2):107–124, 2011. https://potassco.org/ Accessed: July 28, 2024. 5

[GKKS22] Martin Gebser, Roland Kaminski, Benjamin Kaufmann, and Torsten Schaub. *Answer set solving in practice*. Springer Nature, 2022. 6, 14

[GLP05] Michael Genesereth, Nathaniel Love, and Barney Pell. General game playing: Overview of the aaai competition. *AI magazine*, 26(2):62–62, 2005. 14, 40

[Gre18] Elliot Greenwood. Learning player strategies using weak constraints. Master's thesis, Imperial College london, Department of Computing, London, 2018. https://www.imperial.ac.uk/media/imperial-college/faculty-of-engineering/computing/public/1718-ug-projects/Elliot-Greenwood-Learning-Player-Strategies-using-Weak-Constraints.pdf Acessed: July 18, 2025. 15

[Law18] Mark Law. *Inductive learning of answer set programs*. PhD thesis, Imperial College London, UK, 2018. 14

[LHH+08] Nathaniel Love, Timothy Hinrichs, David Haley, Eric Schkufza, and Michael Genesereth. General game playing: Game description language specification, 2008. 14, 16, 17, 40

[Lif19] Vladimir Lifschitz. *Answer Set Programming (Draft)*. Unpublished, 2019. 5, 14

[LRB15] Mark Law, Alessandra Russo, and Krysia Broda. The ILASP system for learning answer set programs. www.ilasp.com, 2015. 5, 11, 14

[MDR94] Stephen Muggleton and Luc De Raedt. Inductive logic programming: Theory and methods. *The Journal of Logic Programming*, 19:629–679, 1994. 5, 14, 15

[SH22] Atreya Shankar Susana Hahn. Investigating the utility of answer set programming and inductive logic techniques in learning two-player game strategies, 2022. 15, 35

[Str24] Myrthe Streep. Learning to play with inductive logic programming. Bachelor's thesis, Open Universiteit, Heerlen, 2024. 5, 19, 20, 33, 40

A. Connect-Four - Reference Program

A.1. Program Listing

```
 c4_asp_v3.lp
% ASP-code for playing Connect Four.
% Version to be used in the learning process

% === P R O G R A M ======== 2025-07-26 ======================================
board_size(7,6).
role(1). role(2).

col(X) :- X = 1..M, board_size(M,N).
row(Y) :- Y = 1..N, board_size(M,N).

% Initial state ---------------------------------------------------------------
init(X,Y,0) :- col(X), row(Y).
init(control(1)).

% Initializing a game -- Not to be used when an explicit game state is given -
% cell(X,Y,P) :- init(X,Y,P).
% control(P) :- init(control(P)).

% Legal moves: Actions --------------------------------------------------------
legal(P, drop(X)) :- control(P), open(X), not terminal.

% Action selection ------------------------------------------------------------
{does(X, A): legal(X, A)} = 1 :- not terminal.

% Game state update  ----------------------------------------------------------
target(X,Y) :- cell(X,Y,0), not cell(X,Y-1,0).
target(X,Y,P) :- does(P, drop(X)), cell(X,Y,0), not cell(X,Y-1,0).

next(X,Y,P) :- cell(X,Y,P), role(P).
next(X,Y,P) :- target(X,Y,P).
next(X,Y,0) :- cell(X,Y,0), not target(X,Y,_).

next(control(2)) :- control(1).
next(control(1)) :- control(2).

% Goal ------------------------------------------------------------------------
goal(P,100) :- line(P).
goal(P, 50) :- not goal(P,100), not goal(P,0), role(P).
goal(P,  0) :- goal(P2,100), P!=P2, role(P).

% Termination -----------------------------------------------------------------
terminal :- line(P).
terminal :- not open.

% Additional concepts ---------------------------------------------------------
open(X) :- cell(X,N,0), board_size(M,N).
open :- open(X).
opponent(1,2). opponent(2,1).

horz(X,Y,P) :- cell(X,Y,P),cell(X+1,Y,P),cell(X+2,Y,P),cell(X+3,Y,P),role(P).
vert(X,Y,P) :- cell(X,Y,P),cell(X,Y+1,P),cell(X,Y+2,P),cell(X,Y+3,P),role(P).
dia1(X,Y,P) :- cell(X,Y,P),cell(X+1,Y+1,P),cell(X+2,Y+2,P),cell(X+3,Y+3,P),role(P).
```

```
54 dia2(X,Y,P) :- cell(X,Y,P),cell(X+1,Y-1,P),cell(X+2,Y-2,P),cell(X+3,Y-3,P),role(P).
55
56 line(P) :- horz(X,Y,P).
57 line(P) :- vert(X,Y,P).
58 line(P) :- dia1(X,Y,P).
59 line(P) :- dia2(X,Y,P).
60
61 % Strategies ------------------------------------------------------------------
62 % Vertical
63 check(P,X) :- cell(X,Y,P), cell(X,Y+1,P), cell(X,Y+2,P), cell(X,Y+3,0). % Just one option
64
65 % Horizontal
66 check(P,X) :- cell(X,Y,0), cell(X+1,Y,P), cell(X+2,Y,P), cell(X+3,Y,P), target(X,Y). %
       .ooo
67 check(P,X) :- cell(X-1,Y,P), cell(X,Y,0), cell(X+1,Y,P), cell(X+2,Y,P), target(X,Y). %
       o.oo
68 check(P,X) :- cell(X-2,Y,P), cell(X-1,Y,P), cell(X,Y,0), cell(X+1,Y,P), target(X,Y). %
       oo.o
69 check(P,X) :- cell(X-3,Y,P), cell(X-2,Y,P), cell(X-1,Y,P), cell(X,Y,0), target(X,Y). %
       ooo.
70
71 % diagonal SW-NE
72 check(P,X) :- cell(X,Y,0), cell(X+1,Y+1,P), cell(X+2,Y+2,P), cell(X+3,Y+3,P),
       target(X,Y).
73 check(P,X) :- cell(X-1,Y-1,P), cell(X,Y,0), cell(X+1,Y+1,P), cell(X+2,Y+2,P),
       target(X,Y).
74 check(P,X) :- cell(X-2,Y-2,P), cell(X-1,Y-1,P), cell(X,Y,0), cell(X+1,Y+1,P),
       target(X,Y).
75 check(P,X) :- cell(X-3,Y-3,P), cell(X-2,Y-2,P), cell(X-1,Y-1,P), cell(X,Y,0),
       target(X,Y).
76
77 % diagonal NW-SE
78 check(P,X) :- cell(X,Y,0),     cell(X+1,Y-1,P),  cell(X+2,Y-2,P), cell(X+3,Y-3,P),
       target(X,Y).
79 check(P,X) :- cell(X-1,Y+1,P), cell(X,Y,0),      cell(X+1,Y-1,P), cell(X+2,Y-2,P),
       target(X,Y).
80 check(P,X) :- cell(X-2,Y+2,P), cell(X-1,Y+1,P),  cell(X,Y,0),     cell(X+1,Y-1,P),
       target(X,Y).
81 check(P,X) :- cell(X-3,Y+3,P), cell(X-2,Y+2,P),  cell(X-1,Y+1,P), cell(X,Y,0),
       target(X,Y).
82
83 % ================================================================================
84
85 % Game play state example
86 cell(1,6,0). cell(2,6,0). cell(3,6,0). cell(4,6,0). cell(5,6,0). cell(6,6,0).
       cell(7,6,0).
87 cell(1,5,0). cell(2,5,0). cell(3,5,0). cell(4,5,0). cell(5,5,0). cell(6,5,0).
       cell(7,5,0).
88 cell(1,4,0). cell(2,4,0). cell(3,4,0). cell(4,4,0). cell(5,4,0). cell(6,4,0).
       cell(7,4,0).
89 cell(1,3,0). cell(2,3,0). cell(3,3,1). cell(4,3,0). cell(5,3,0). cell(6,3,0).
       cell(7,3,0).
90 cell(1,2,0). cell(2,2,0). cell(3,2,1). cell(4,2,0). cell(5,2,0). cell(6,2,0).
       cell(7,2,0).
91 cell(1,1,0). cell(2,1,2). cell(3,1,1). cell(4,1,2). cell(5,1,2). cell(6,1,0).
       cell(7,1,0).
92 control(1).
93
```

```
% Game text scheme example
% . . . . . . .
% . . . . . . .
% . . . . . . .
% . . 1 . . . .
% . . 1 . . . .
% . 2 1 2 2 . .

% Directives ------------------------------------------------------------------------
#show next/3.

% clingo version 5.4.0
% Reading from ...ers/boschloo/connect4/asp/c4_asp_v3.lp
% Solving...
% Answer: 1
% next(1,6,0) next(2,6,0) next(3,6,0) next(4,6,0) next(5,6,0) next(6,6,0) next(7,6,0)
    next(1,5,0) next(2,5,0) next(3,5,0) next(4,5,0) next(5,5,0) next(6,5,0) next(7,5,0)
    next(1,4,0) next(2,4,0) next(3,4,0) next(4,4,0) next(5,4,0) next(6,4,0) next(7,4,0)
    next(1,3,0) next(2,3,0) next(4,3,0) next(5,3,0) next(6,3,0) next(7,3,0) next(1,2,0)
    next(2,2,0) next(4,2,0) next(5,2,0) next(6,2,0) next(7,2,0) next(3,2,0) next(3,3,0)
    next(3,1,1) next(3,3,1) next(3,2,1) next(2,1,2) next(4,1,2) next(5,1,2) next(6,1,0)
    next(7,1,0) next(2,1,0) next(3,1,0) next(4,1,0) next(5,1,0) next(1,1,1)
% Answer: 2
% next(1,6,0) next(2,6,0) next(3,6,0) next(4,6,0) next(5,6,0) next(6,6,0) next(7,6,0)
    next(1,5,0) next(2,5,0) next(3,5,0) next(4,5,0) next(5,5,0) next(6,5,0) next(7,5,0)
    next(1,4,0) next(2,4,0) next(3,4,0) next(4,4,0) next(5,4,0) next(6,4,0) next(7,4,0)
    next(1,3,0) next(2,3,0) next(4,3,0) next(5,3,0) next(6,3,0) next(7,3,0) next(1,2,0)
    next(2,2,0) next(4,2,0) next(5,2,0) next(6,2,0) next(7,2,0) next(3,2,0) next(3,3,0)
    next(3,1,1) next(3,3,1) next(3,2,1) next(2,1,2) next(4,1,2) next(5,1,2) next(1,1,0)
    next(6,1,0) next(7,1,0) next(3,1,0) next(4,1,0) next(5,1,0) next(2,1,1)
% Answer: 3
% next(1,6,0) next(2,6,0) next(3,6,0) next(4,6,0) next(5,6,0) next(6,6,0) next(7,6,0)
    next(1,5,0) next(2,5,0) next(3,5,0) next(4,5,0) next(5,5,0) next(6,5,0) next(7,5,0)
    next(1,4,0) next(2,4,0) next(3,4,0) next(4,4,0) next(5,4,0) next(6,4,0) next(7,4,0)
    next(1,3,0) next(2,3,0) next(4,3,0) next(5,3,0) next(6,3,0) next(7,3,0) next(1,2,0)
    next(2,2,0) next(4,2,0) next(5,2,0) next(6,2,0) next(7,2,0) next(3,2,0) next(3,3,0)
    next(3,1,1) next(3,3,1) next(3,2,1) next(2,1,2) next(4,1,2) next(5,1,2) next(1,1,0)
    next(6,1,0) next(7,1,0) next(2,1,0) next(4,1,0) next(5,1,0)
% Answer: 4
% next(1,6,0) next(2,6,0) next(3,6,0) next(4,6,0) next(5,6,0) next(6,6,0) next(7,6,0)
    next(1,5,0) next(2,5,0) next(3,5,0) next(4,5,0) next(5,5,0) next(6,5,0) next(7,5,0)
    next(1,4,0) next(2,4,0) next(3,4,0) next(4,4,0) next(5,4,0) next(6,4,0) next(7,4,0)
    next(1,3,0) next(2,3,0) next(4,3,0) next(5,3,0) next(6,3,0) next(7,3,0) next(1,2,0)
    next(2,2,0) next(4,2,0) next(5,2,0) next(6,2,0) next(7,2,0) next(3,2,0) next(3,3,0)
    next(3,1,1) next(3,3,1) next(3,2,1) next(2,1,2) next(4,1,2) next(5,1,2) next(1,1,0)
    next(6,1,0) next(7,1,0) next(2,1,0) next(3,1,0) next(5,1,0) next(4,1,1)
% Answer: 5
% next(1,6,0) next(2,6,0) next(3,6,0) next(4,6,0) next(5,6,0) next(6,6,0) next(7,6,0)
    next(1,5,0) next(2,5,0) next(3,5,0) next(4,5,0) next(5,5,0) next(6,5,0) next(7,5,0)
    next(1,4,0) next(2,4,0) next(3,4,0) next(4,4,0) next(5,4,0) next(6,4,0) next(7,4,0)
    next(1,3,0) next(2,3,0) next(4,3,0) next(5,3,0) next(6,3,0) next(7,3,0) next(1,2,0)
    next(2,2,0) next(4,2,0) next(5,2,0) next(6,2,0) next(7,2,0) next(3,2,0) next(3,3,0)
    next(3,1,1) next(3,3,1) next(3,2,1) next(2,1,2) next(4,1,2) next(5,1,2) next(1,1,0)
    next(6,1,0) next(7,1,0) next(2,1,0) next(3,1,0) next(4,1,0) next(5,1,1)
% Answer: 6
% next(1,6,0) next(2,6,0) next(3,6,0) next(4,6,0) next(5,6,0) next(6,6,0) next(7,6,0)
    next(1,5,0) next(2,5,0) next(3,5,0) next(4,5,0) next(5,5,0) next(6,5,0) next(7,5,0)
    next(1,4,0) next(2,4,0) next(3,4,0) next(4,4,0) next(5,4,0) next(6,4,0) next(7,4,0)
```

```
170      next(1,3,0) next(2,3,0) next(4,3,0) next(5,3,0) next(6,3,0) next(7,3,0) next(1,2,0)
171      next(2,2,0) next(4,2,0) next(5,2,0) next(6,2,0) next(7,2,0) next(3,2,0) next(3,3,0)
172      next(3,1,1) next(3,3,1) next(3,2,1) next(2,1,2) next(4,1,2) next(5,1,2) next(1,1,0)
173      next(6,1,0) next(2,1,0) next(3,1,0) next(4,1,0) next(5,1,0) next(7,1,1)
174  % Answer: 7
175  % next(1,6,0) next(2,6,0) next(3,6,0) next(4,6,0) next(5,6,0) next(6,6,0) next(7,6,0)
176      next(1,5,0) next(2,5,0) next(3,5,0) next(4,5,0) next(5,5,0) next(6,5,0) next(7,5,0)
177      next(1,4,0) next(2,4,0) next(3,4,0) next(4,4,0) next(5,4,0) next(6,4,0) next(7,4,0)
178      next(1,3,0) next(2,3,0) next(4,3,0) next(5,3,0) next(6,3,0) next(7,3,0) next(1,2,0)
179      next(2,2,0) next(4,2,0) next(5,2,0) next(6,2,0) next(7,2,0) next(3,2,0) next(3,3,0)
180      next(3,1,1) next(3,3,1) next(3,2,1) next(2,1,2) next(4,1,2) next(5,1,2) next(1,1,0)
181      next(7,1,0) next(2,1,0) next(3,1,0) next(4,1,0) next(5,1,0) next(6,1,1)
182  % SATISFIABLE
183
184  % Models       : 7
185  % Calls        : 1
186  % Time         : 0.014s (Solving: 0.00s 1st Model: 0.00s Unsat: 0.00s)
187  % CPU Time     : 0.006s
```

Listing 24: ASP reference program of Connect-Four

A.2. EXPLANATION

The code of Connect-Four is composed using examples of other games in [GLP05], [LHH+08]. The code is created in parallel with the code for Go [Str24]. In the ASP-version the GDL guidelines are loosely followed. A sample of the output is added. Line numbers refer to the program listing.

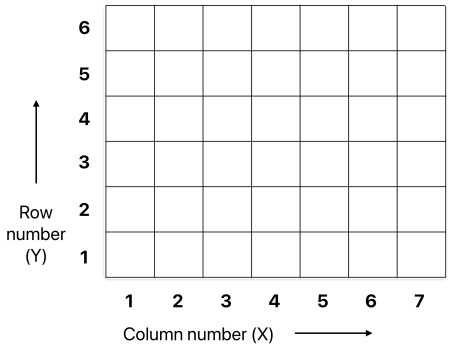

Figure 5: Connect-Four Game Board with coordinates

- **Game Board** The board is modeled as a grid with coordinates X and Y. This gives cell/3 predicates, like cell(X,Y,P). The X-coordinate counts the columns left to right and the Y-coordinate counts the rows from bottom to top. The P ("player") indicates the content of the cell. Initially, all cells are empty, indicated with 0.

At the top of the program, the dimensions of the board are given as a fact in board_size/2.

- **Role** In GDL, players are defined with the role/1 relation. In this Connect-Four code, the players are indicated by 1 for white and 2 for black. Players are designated by numbers instead of white and black. This shortens the program lines and answer sets and reduces the chance of errors due to the similarity between 'black' and 'blank'. The terms player and role are used interchangeably.
- **Init** Init describes the situation at the start of the game. All fields are empty and get the value 0. White (role 1) always makes the first move. When the game is played by a computer, the control program takes the initial state as the actual state. For now, the init part is ignored. For each test, we provide true/1 predicates with the actual state.
- **True** The actual state of the game: board, sometimes the player in control. True/1 is a given for all game play situations. In test cases, this will be the input. In the ASP-version this is shortened to just cell/3.
- **Legal** The predicate legal(Player,Move) sums up the actions that a player can do in different game states. In Connect-Four the player in control can drop a piece as long as the game is not terminal. In literature, sometimes a 'noop' action is added for the player whose turn it is not. Unlike in some other games, a player in Connect-Four is not allowed to skip a turn.
- **Actions** does(Player,Move) indicates the action a player actually performs when it is her turn. A choice is made from the legal possible moves. In Connect-Four this is a drop/1 in one of the seven columns, or less, if columns are filled up already.
- **Next** Next/1 defines the state of board and players after a player has dropped a disk in column X. All fields with disks remain unchanged (line 30), only the lowest empty field in the column X gets another 'color' (31). Empty fields get the color 0 (32). Furthermore, the control changes from one player to the other (34, 35).
- **Goal** goal/2 indicates a winning situation for one of the players. This predicate is included to be consistent with the GDL-mindset.
- **Terminal** A player succeeding in making a line of four disks wins and the game ends. The additional predicate line/1 is true. The game also ends when all 42 pieces have been played and no line has been created. The predicate open/1 is true when there is still room for at least one more disc. Predicate open/0 is true when there is still a playable position in one of the columns. These predicate can be used to determine terminal/0.

- **Line** is the essential part of Connect-Four. A line of four discs can be in one of four directions: horizontal, vertical, and two flavors diagonally. For different directions, different names (horz/3, vert/3, diag1/3, diag2/3) are used.
- **Check** The check rules are added to assist in learning strategies. An explanation can be found in paragraph 4.1.4.

B. CONNECT-FOUR - LEARNED PROGRAM

```
% c4_asp_LEARNED.lp
%
% Version compiled from the learned rules

% Background  given ============================
role(1). role(2).
col(X) :- cell(X,_,_).
row(Y) :- cell(_,Y,_).

% NumberOf/2 given -----------------------------
count(1,0,P,0) :- role(P).
count(X,Y,P,N) :- count(X,Y-1,P,N-1), cell(X,Y,P).
count(X,1,P,N) :- count(X-1,Y-1,P,N-1), not cell(X-1,Y,_), cell(X,1,_), cell(X,1,P).
count(X,Y,P,N) :- count(X,Y-1,P,N), cell(X,Y,_), not cell(X,Y,P).
count(X,1,P,N) :- count(X-1,Y-1,P,N), not cell(X-1,Y,_), cell(X,1,_), not cell(X,1,P).
numberOf(P,N) :- count(X,Y,P,N), board_size(X,Y).

% Board_size v705 ------------------------------
greaterY(V1) :- row(V1); row(V2); V2 > V1.
greaterX(V1) :- col(V1); col(V2); V2 > V1.
board_size(V1,V2) :- col(V1); row(V2); not greaterX(V1); not greaterY(V2).

% control v704 ---------------------------------
control(V1) :- numberOf(V1,V3); numberOf(V2,V3); V1 < V2.
control(V1) :- numberOf(V1,V2); numberOf(V4,V3); V2 < V3.

% Legal v7071 ----------------------------------
legal(V1,drop(V2)) :- control(V1); open(V2); not terminal.

% Action selection v719  =======================
1 {does(V2,V1) : legal(V2,V1) } 1.

% target/2 v706 --------------------------------
target(V1,V2) :- cell(V1,V2,0); not cell(V1,V2-1,0).
target(V1,V2,V3) :- target(V1,V2); does(V3,drop(V1)).

% next v710, v718 ------------------------------
next(V1,V2,V3) :- target(V1,V2,V3).
next(V1,V2,V3) :- role(V3); cell(V1,V2,V3).
next(V1,V2,0) :- control(V3); cell(V1,V2,0); not target(V1,V2,V3).
next(control(V1)) :- opponent(V2,V1); not next(control(V2)).

% goal v713 ------------------------------------
goal(V1,100) :- line(V1).
goal(V1,50) :- opponent(V1,V2); not terminal.
goal(V1,0) :- line(V2); opponent(V1,V2).

% terminal v715 --------------------------------
terminal :- line(V1).
terminal :- not open.

% open v717 ------------------------------------
open(V1) :- cell(V1,V2,0).
open :- cell(V1,V2,0).

% opponent v708c -------------------------------
```

```
opponent(V1,V2) :- role(V1); role(V2); V2 != V1.

% Row-of-four v700, v701, v702, v703-------------
horz(V1,V2,V3) :- role(V3); cell(V1,V2,V3); cell(V1+1,V2,V3); cell(V1+2,V2,V3);
    cell(V1+3,V2,V3).
vert(V1,V2,V3) :- role(V3); cell(V1,V2,V3); cell(V1,V2+1,V3); cell(V1,V2+2,V3);
    cell(V1,V2+3,V3).
dia1(V1,V2,V3) :- role(V3); cell(V1,V2,V3); cell(V1+1,V2+1,V3); cell(V1+2,V2+2,V3);
    cell(V1+3,V2+3,V3).
dia2(V1,V2,V3) :- role(V3); cell(V1,V2,V3); cell(V1+1,V2-1,V3); cell(V1+2,V2-2,V3);
    cell(V1+3,V2-3,V3).

% line v711 ------------------------------------
line(V1) :- horz(V2,V3,V1).
line(V1) :- vert(V2,V3,V1).
line(V1) :- dia1(V2,V3,V1).
line(V1) :- dia2(V2,V3,V1).

% Checks given -----------------------------
% Vertical
check(P,X) :- cell(X,Y,P), cell(X,Y+1,P), cell(X,Y+2,P), cell(X,Y+3,0). % Just one option

% Horizontal
check(P,X) :- cell(X,Y,0), cell(X+1,Y,P), cell(X+2,Y,P), cell(X+3,Y,P), target(X,Y). %
    .ooo
check(P,X) :- cell(X-1,Y,P), cell(X,Y,0), cell(X+1,Y,P), cell(X+2,Y,P), target(X,Y). %
    o.oo
check(P,X) :- cell(X-2,Y,P), cell(X-1,Y,P), cell(X,Y,0), cell(X+1,Y,P), target(X,Y). %
    oo.o
check(P,X) :- cell(X-3,Y,P), cell(X-2,Y,P), cell(X-1,Y,P), cell(X,Y,0), target(X,Y). %
    ooo.

% diagonal SW-NE
check(P,X) :- cell(X,Y,0), cell(X+1,Y+1,P), cell(X+2,Y+2,P), cell(X+3,Y+3,P),
    target(X,Y).
check(P,X) :- cell(X-1,Y-1,P), cell(X,Y,0), cell(X+1,Y+1,P), cell(X+2,Y+2,P),
    target(X,Y).
check(P,X) :- cell(X-2,Y-2,P), cell(X-1,Y-1,P), cell(X,Y,0), cell(X+1,Y+1,P),
    target(X,Y).
check(P,X) :- cell(X-3,Y-3,P), cell(X-2,Y-2,P), cell(X-1,Y-1,P), cell(X,Y,0),
    target(X,Y).

% diagonal NW-SE
check(P,X) :- cell(X,Y,0),     cell(X+1,Y-1,P),  cell(X+2,Y-2,P), cell(X+3,Y-3,P),
    target(X,Y).
check(P,X) :- cell(X-1,Y+1,P), cell(X,Y,0),      cell(X+1,Y-1,P), cell(X+2,Y-2,P),
    target(X,Y).
check(P,X) :- cell(X-2,Y+2,P), cell(X-1,Y+1,P),  cell(X,Y,0),     cell(X+1,Y-1,P),
    target(X,Y).
check(P,X) :- cell(X-3,Y+3,P), cell(X-2,Y+2,P),  cell(X-1,Y+1,P), cell(X,Y,0),
    target(X,Y).

% preferences v801 -----------------------------
:~ does(V1,drop(V2)), not center(V2).[20@1, V1, V2]
:~ does(V1,drop(V2)), not check(V1,V2).[20@2, V1, V2]
:~ does(V1,drop(V2)), role(V3), not check(V3,V2).[20@3, V1, V2, V3]

center(X) :- X > 2, X < 6, col(X).
```

```
115
116 % Game status ---------------------------------------------------------------------------
117 cell(1,6,0). cell(2,6,0). cell(3,6,0). cell(4,6,0). cell(5,6,0). cell(6,6,0).
118     cell(7,6,0).
119 cell(1,5,0). cell(2,5,0). cell(3,5,0). cell(4,5,2). cell(5,5,0). cell(6,5,0).
120     cell(7,5,0).
121 cell(1,4,0). cell(2,4,0). cell(3,4,0). cell(4,4,1). cell(5,4,1). cell(6,4,0).
122     cell(7,4,0).
123 cell(1,3,2). cell(2,3,2). cell(3,3,0). cell(4,3,2). cell(5,3,2). cell(6,3,0).
124     cell(7,3,0).
125 cell(1,2,1). cell(2,2,1). cell(3,2,0). cell(4,2,1). cell(5,2,1). cell(6,2,1).
126     cell(7,2,2).
127 cell(1,1,2). cell(2,1,1). cell(3,1,1). cell(4,1,2). cell(5,1,2). cell(6,1,1).
128     cell(7,1,2).
129
130 % . . . . . . .
131 % . . . 2 . . .
132 % . . . 1 1 . .
133 % 2 2 . 2 2 . .
134 % 1 1 . 1 1 1 2
135 % 2 1 1 2 2 1 2
136
137 #show next/3.
```

Listing 25: With ILASP learned program of Connect-Four

C. CONNECT-FOUR - SIMULATOR

A web app was made for playing Connect-Four, replaying examples from literature and generating, storing and documenting played games.

This app consists of an index.html file, with css styling and a Javascript program. It can be used locally, without a web server running. The working app can be found on https://kladblok.app

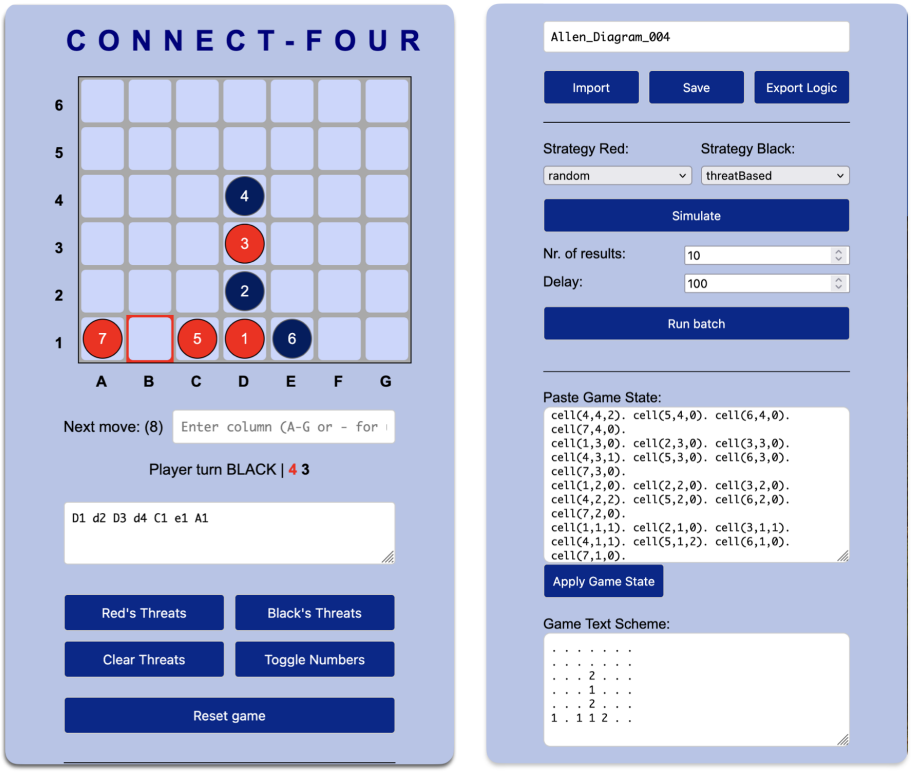

Figure 6: Connect-Four Simulator

- Moves can be made by clicking on a column or entering a character `A-G` in the first input window. A move can be withdrawn by typing a minus sign.
- The active player is shown below the input window.
- Game play is recorded in a note pad window. Moves are noted according to [All10].
- Move numbers can be shown or hidden with the Toggle numbers button.
- You have options to show threats for both players with thin red or black frames.

- The Reset Game button restart the game without reloading the page.
- Import loads a game from disc, Save saves a complete game with all the moves in the correct order as a json file. Without a name entered, a default name is used. Older versions will not be overwritten.
- To export a game state as text, Export Logic is used. The format is specific for use in ILASP.
- The other way round, you can paste ILASP output (`cell/3, next/3`) in the Paste Game State window. On clicking Apply Game State, the board becomes visible and the Game Text Scheme below is being updated.
- To generate a game using a built-in strategy, choose the desired strategy for both players and click on Simulate. Some basic strategies are available.
- A batch function (Run Batch) creates a given number of simulation results.