Knowledge Representation and Learning UE (351.010)
Assignment 2

Giovanni Filomeno - K12315325
Manu Gupta - K11936702
Sarah Lackner - K11907930

June 9, 2024

In the following report we shortly describe the Aleph and Popper encodings for each exercise and provide remarks on the process. Both Aleph and Popper are inductive logic programming (ILP) systems used to learn rules from examples.

Exercise 1

Task 1: Trains

This expercise involves learning rules to identify eastbound trains using two Inductive Logic Programming (ILP) systems, Aleph and Popper. We provided both systems with positive and negative examples of eastbound trains and background knowledge about the train's properties.

Aleph Setup: For Aleph, the files for positive examples (`train.f`), negative examples (`train.n`), and background knowledge (`train.b`) were already avaailable. The mode declarations and type definitions guided Aleph's hypothesis generation.

Popper Setup For Popper, we defined background knowledge (`bk.pl`), bias (`bias.pl`), and examples (`exs.pl`) using information from the files in Aleph setup. The bias file constrained the search space, defining the head and body predicates and their types.

Positive Examples The positive examples for the train problem indicate trains that are eastbound:

```
% Positive examples
eastbound(east1).
eastbound(east2).
eastbound(east3).
eastbound(east4).
eastbound(east5).
```

Negative Examples: Negative examples identify trains that are not eastbound:

```
% Negative examples
eastbound(west6).
eastbound(west7).
eastbound(west8).
eastbound(west9).
eastbound(west10).
```

Execution: We ran Aleph and Popper using scripts to automate the induction process and recorded the results. Aleph was executed in a Prolog environment, while Popper was run using Python scripts interfacing with its API.

Aleph's Solution

Output:

```
[theory]

[Rule 1] [Pos cover = 5 Neg cover = 0]
eastbound(A) :-
   has_car(A,B), short(B), closed(B).

[Training set performance]
          Actual
        +          -
     +  5          0          5
Pred
     -  0          5          5

        5          5         10

Accuracy = 1
[Training set summary] [[5,0,0,5]]
```

Process:

- Aleph used a bottom-up approach, generating a bottom clause with all possible literals.
- It constructed and reduced clauses to find the best hypothesis.
- Generated 70 clauses, selected the best clause covering all positive examples with no false positives or negatives.
- The final rule learned: `eastbound(A) :- has_car(A,B), short(B), closed(B).`
- **Accuracy:** 100%
- **Total Clauses Constructed:** 70
- **Total Time Taken:** 0.011 seconds

Popper's Solution

Output:

```
10:40:08 Generating programs of size: 2
10:40:08 Generating programs of size: 3
10:40:08 Generating programs of size: 4
********** SOLUTION **********
Precision:1.00 Recall:1.00 TP:5 FN:0 TN:5 FP:0 Size:4
eastbound(V0):- has_car(V0,V1), short(V1), closed(V1).
*****************************
Num. programs: 46
Total operation time: 0.16s
Total execution time: 0.17s
```

Process:

- Popper used a top-down approach, iteratively generating and refining hypotheses.
- It evaluated hypotheses by testing their precision and recall.
- Generated 46 programs to find the best hypothesis.
- The final rule learned: `eastbound(V0):- has_car(V0,V1), short(V1), closed(V1).`
- **Accuracy:** 100%
- **Total Programs Generated:** 46
- **Total Time Taken:** 0.17 seconds

Comparative Analysis

- **Hypotheses Generated**
  - **Aleph:** Generated and evaluated 70 clauses.
  - **Popper:** Generated and evaluated 46 programs.
- **Complexity of Solution**
  - Both Aleph and Popper arrived at the same final rule: `eastbound(X) :- has_car(X, Y), short(Y), closed(Y).`
  - The rule is concise and effectively distinguishes between eastbound and non-eastbound trains.
- **Performance**
  - **Aleph:** Constructed 70 clauses in 0.011 seconds.
  - **Popper:** Generated 46 programs in 0.17 seconds.
  - Popper's approach took longer due to the iterative refinement process but generated fewer hypotheses.
- **Accuracy**
  - Both systems achieved 100% accuracy, correctly classifying all positive and negative examples.

Summary Both Aleph and Popper effectively learned the correct hypothesis for the train problem, achieving perfect accuracy. Aleph generated more clauses in a shorter amount of time using a bottom-up approach, while Popper generated fewer programs but took longer due to its top-down approach. Despite these differences, both systems arrived at the same concise and accurate rule, demonstrating their capability to solve the problem effectively.

Task 2: Zendo Game

Experimental Setup We applied Aleph and Popper to learn rules for the Zendo game, where the goal is to identify structures following a specific pattern. Positive and negative examples were provided, along with background knowledge about the pieces' properties.

Aleph Setup For Aleph, we created files for background knowledge (`zendo.b`), positive examples (`zendo.f`), and negative examples (`zendo.n`), along with mode declarations and types.

Popper Setup For Popper, we defined files for background knowledge (`bk.pl`), bias (`bias.pl`), and examples (`exs.pl`). Popper's bias files defined the search space constraints.

Positive Examples Positive examples for the Zendo game indicate structures that follow the pattern:

```
% Positive examples
pos(zendo(p1)).
pos(zendo(p2)).
pos(zendo(p3)).
pos(zendo(p4)).
```

Negative Examples Negative examples identify structures that do not follow the pattern:

```
% Negative examples
neg(zendo(p5)).
neg(zendo(p6)).
neg(zendo(p7)).
neg(zendo(p8)).
```

Execution Both tasks were executed by running Aleph and Popper scripts. Popper was run using a Python script managing execution environments and outputs, while Aleph was executed in Prolog.

Aleph's Solution

Output:

```
[theory]

[Rule 1] [Pos cover = 1 Neg cover = 0]
zendo(p1).

[Rule 2] [Pos cover = 1 Neg cover = 0]
zendo(p2).

[Rule 3] [Pos cover = 1 Neg cover = 0]
zendo(p3).

[Rule 4] [Pos cover = 1 Neg cover = 0]
zendo(p4).

[Training set performance]
         Actual
       +        -
     + 4        0        4
Pred
     - 0        4        4

       4        4        8

Accuracy = 1
[Training set summary] [[4,0,0,4]]
[time taken] [0.0021010000000000195]
% [total clauses constructed] [0]
% true.
```

Process:

- Aleph used a bottom-up approach, generating a bottom clause with all possible literals.
- It constructed and reduced clauses to find the best hypothesis.
- Selected the best clause covering all positive examples with no false positives or negatives.
- **Accuracy:** 100%
- **Total Time Taken:** 0.0021 seconds

Popper's Solution

Output:

```
12:49:01 Generating programs of size: 2
12:49:01 Generating programs of size: 3
12:49:01 Generating programs of size: 4
12:49:01 Generating programs of size: 5
********** SOLUTION **********
Precision:1.00 Recall:1.00 TP:20 FN:0 TN:20 FP:0 Size:6
zendo(V0):- small(V2),piece(V0,V1),red(V1),contact(V1,V3),size(V3,V2).
*****************************
Num. programs: 1412
Total operation time: 0.68s
Total execution time: 0.71s
```

Process:

- Popper used a top-down approach, iteratively generating and refining hypotheses.
- It evaluated hypotheses by testing their precision and recall.
- Generated 1412 programs to find the best hypothesis.
- The final rule learned: `zendo(V0):- small(V2),piece(V0,V1),red(V1),contact(V1,V3),size(V3,V2).`
- **Accuracy:** 100%
- **Total Programs Generated:** 1412
- **Total Time Taken:** 0.71 seconds

Comparative Analysis

- **Hypotheses Generated:**
  - **Aleph:** Generated and evaluated 0 clauses.
  - **Popper:** Generated and evaluated 1412 programs.
- **Complexity of Solution:**
  - Aleph learned individual rules for each positive example.
  - Popper derived a single, more complex rule that generalizes over all positive examples.
- **Performance:**
  - **Aleph:** Constructed 0 clauses in 0.0021 seconds.
  - **Popper:** Generated 1412 programs in 0.71 seconds.
  - Popper's approach took significantly longer due to the iterative refinement process but generated a general rule covering all positive examples.
- **Accuracy:**
  - Both systems achieved 100% accuracy, correctly classifying all positive and negative examples.

Summary For the Zendo game problem, Popper generated a complex hypothesis that achieved perfect precision and recall, effectively covering all 20 positive and 20 negative examples with a final rule of size 6: `zendo(V0):- small(V1), piece(V0, V2), red(V2), contact(V2, V3), size(V3, V1)`. This rule combined multiple conditions (`small`, `piece`, `red`, `contact`, `size`) to determine valid Zendo examples. Conversely, Aleph, generated overly specific rules that do not generalize well. Each rule simply identifies one of the training examples as positive without extracting a meaningful pattern. On the other hand, Popper successfully generated a generalized rule that captures the underlying pattern in the positive examples. This highlights Popper's strength in producing more useful and generalized hypotheses in the context of the Zendo game problem.

Exercise 2

Experimental Setup In this exercise, we applied the two ILP systems, i.e., Aleph and Popper, to learn recursive definitions for two list-processing tasks: finding the maximum element (`max`) and calculating the sum (`sum`) of elements in a list. We defined positive and negative examples, as well as background knowledge for each task, tailored specifically to the requirements of each ILP system.

Aleph Setup For Aleph, we created separate files for background knowledge (`bk`), examples (`f` for positive and `n` for negative), and mode declarations (`b`). These files specify the predicates, types, and directions that guide Aleph's hypothesis generation process.

```
max_assignment([10], 10).
max_assignment([10,10,5], 10).
max_assignment([7,3,9,2], 9).
max_assignment([-1, -3, -2, -1], -1).
max_assignment([0, -1, -2, -3], 0).
max_assignment([50, 20, 30, 40], 50).
max_assignment([2, 5, 3, 5, 2], 5).
max_assignment([12, 15, 14, 13], 15).
max_assignment([21], 21).
max_assignment([99, 101, 100], 101).
max_assignment([6, 6, 6, 6], 6).
max_assignment([-5, 1, -1, 0], 1).
```

Figure 1: Max positive examples

```
sum_assignment([4,2,6], 12).
sum_assignment([1,1,1], 3).
sum_assignment([3,5], 8).
sum_assignment([], 0).
sum_assignment([10,10,5], 25).
sum_assignment([3,4,5], 12).
sum_assignment([1, 2, 3, 4, 5], 15).
```

Figure 2: Sum positive examples

Popper Setup For Popper, we defined files for background knowledge (`bk.pl`), bias (`bias.pl`), and examples (`exs.pl`). Popper's bias files are crucial as they define the search space by setting constraints on the maximum variables, clauses, and the types of predicates allowed in the hypothesis.

Positive Examples Positive examples are designed to represent correct applications of the predicates we are trying to learn. For instance:

- For the `sum` task: `pos(sum([1, 2, 3], 6))` asserts that the sum of the list [1, 2, 3] is 6.
- For the `max` task: `pos(max([1, 2, 3, 4], 4))` asserts that the maximum element of the list [1, 2, 3, 4] is 4.

Negative Examples Negative examples are used to delineate the boundaries of what the predicate should not learn. For example:

- For the `sum` task: `neg(sum([1, 2, 3], 5))` indicates that 5 is not the correct sum of the elements in the list [1, 2, 3].
- For the `max` task: `neg(max([1, 2, 3, 4], 3))` helps ensure that the ILP system does not incorrectly learn that 3 is the maximum when it is not.

These examples help ILP systems understand not only what is true but also what is false, which is crucial for defining accurate and robust rules.

Execution The tasks were executed by running Aleph and Popper through scripts that automate the induction process and record results. Popper was run using a Python script that interfaces with its API to manage execution environments and handle outputs. Aleph was executed through a Prolog script that sets up the environment, loads data, and calls the induction engine.

```
max_assignment([4,2,6], 4).
max_assignment([1,1,1], 0).
max_assignment([3,5], 3).
max_assignment([10], 9).
max_assignment([10,10,5], 5).
max_assignment([7,3,9,2], 7).
max_assignment([-1, -3, -2, -1], -2).
```

Figure 3: Max negative examples

```
sum_assignment([4,2,6], 11).
sum_assignment([1,1,1], 2).
sum_assignment([3,5], 7).
sum_assignment([], 1).
sum_assignment([10,10,5], 24).
sum_assignment([7,3,9,2], 20).
sum_assignment([1, 2, 3, 4, 5], 16).
```

Figure 4: Sum negative examples

Aleph Results Aleph's output for both tasks showed successful rule induction with high accuracy. The rules generated for the `sum` task illustrated typical recursive decompositions and base cases. For the `max` task, Aleph correctly identified the maximum element by considering head and tail operations, along with comparisons to find the greatest element.

Popper Results Popper generated more complex rules for both tasks, reflecting its ability to explore a deeper search space. The recursive structures in Popper's solutions were explicit, with multiple base and recursive cases well integrated into the solutions.

Comparative Analysis Both systems achieved high precision and recall in their tasks, indicating effective learning based on the provided examples. However, Popper's ability to handle recursion was more evident, leading to potentially more robust rules at the cost of increased complexity. In contrast, Aleph's solutions were typically simpler, possibly favoring generalization but at the risk of missing deeper recursive patterns present in the tasks.

The analysis highlights the strengths and limitations of each ILP system: Aleph's efficiency and simplicity versus Popper's depth and complexity.

Exercise 3

We chose the chess KRK (King, Rook, King) endgame - Recognizing illegal positions: https://www.doc.ic.ac.uk/~shm/chess.html
The chess board has 0 to 7 columns and rows respectively. In the provided background.b file the predicate `adj/2` represents column or row values that are adjacent. The `lt/2` encodes column or row values where one is less than the other - which results in an ordering of the columns and rows. The unaltered background knowledge produces a rule for every positive example. Before generalizing anything 3240 rules were learned with and accuracy = 1, meaning there were no misclassifications.

Illegal white-to-move positions

1. Pieces having the same position
2. Black king in check
3. Kings adjacent to each other

Learn a rule set with Aleph In order to obtain a generalization, we added some mode declarations to the train.b file. For a detailed encoding of the modes we refer to the train.b file. Our first approach was to encode each column and row of white king (WK), white rook (WR) and the black king (BK).

- `:- modeh(*, illegal(+colwk, +rowwk, +colwr, +rowwr, +colbk, +rowbk)).` represent what rule we want to learn (head)
- `:- modeb(*, adj(+colwk, +colbk)).`: check if column of WK is adjacent to the column of BK
- `:- modeb(*, lt(+colwk, +colbk)).`: check if column of WK is less than column of BK
- `:- modeb(*, not(lt(+colwk, +colbk))).`: check if column of WK is greater than column of BK

```
[theory]

[Rule 1] [Pos cover = 524 Neg cover = 0]
illegal(A,B,C,D,E,F) :-
   lt(A,E), same(E,C).

[Rule 2] [Pos cover = 1145 Neg cover = 0]
illegal(A,B,C,D,E,F) :-
   adj(A,E), adj(B,F).

[Rule 4] [Pos cover = 525 Neg cover = 0]
illegal(A,B,C,D,E,F) :-
   lt(E,A), same(E,C).

[Rule 5] [Pos cover = 165 Neg cover = 0]
illegal(A,B,C,D,E,F) :-
   same(A,C), same(B,D).

[Rule 6] [Pos cover = 514 Neg cover = 0]
illegal(A,B,C,D,E,F) :-
   lt(B,F), not(lt(D,F)), not(lt(F,D)).

[Rule 8] [Pos cover = 549 Neg cover = 0]
illegal(A,B,C,D,E,F) :-
   lt(F,B), not(lt(D,F)), not(lt(F,D)).

[Rule 11] [Pos cover = 66 Neg cover = 0]
illegal(A,B,C,D,E,F) :-
   adj(C,E), sameb(B,D,F).
```

Figure 5: Rule set Aleph

- `:- modeb(*, same(+colwk, +colwr)).`: checks if WK and WR have the same column (also encoded for rows and BK)
- `:- modeb(*, sameb(+colwk, +colwr, +colbk)).`: checks if WK, WR and BK have the same column (also encoded for rows)
- `:- set(minpos, 20).`: conver at least 20 samples

In order to check if pieces are on the same position (columns or rows), we added the following predicates which check the entries for equality:

- `same(X,Y) :- number(X), number(Y), X == Y.`
- `sameb(X,Y,Z) :- number(X), number(Y), number(Z), X == Y , Y == Z.`

In order to also represent "greater than" rows or columns we can invert the already given facts `not(lt/2)`.

For learning rules with Aleph load the training examples and background knowledge with `read_all(train)` and afterwards run `induce`.. We now get the following rules set:

Figure 5 shows the rule set for determining illegal moves. Due to the restriction to only add rules which cover at least 20 positive examples, some rules are skipped.

Aleph generated `A,B,C,D,E,F` to encode the variables `+colwk, +rowwk, +colwr, +rowwr, +colbk, +rowbk` within the predicate definition.

- `colwk` represent the column of the white king
- `rowwk` represent the row of the white king (WK)
- `wr` represent the white rook (WR)
- `bk` represents the black king (BK)

`[Rule 2] [Pos cover = 1259 Neg cover = 0] illegal(A,B,C,D,E,F) :- adj(A,E), adj(B,F).`
Rule 2 covers 1145 positive examples and encodes the following: A position is illegal if the column index of WK is adjacent to the column index of BK and the row index of the WR is adjacent to the row index of the BK. (See rule 3 - illegal positions)

With the rule set 5 we obtain the result shown in 6:

Verifying found rules manually Let's look at a few of the rules in detail in order to check their validity.

1. `illegal(A,B,C,D,E,F) :- lt(A,E), same(E,C).` This rules indicates that it is illegal if the WK is to the left of the BK and the WR is in the same column as the BK. This makes sense because the BK is in check by the WR, making it an illegal position.

```
[Training set performance]
            Actual
          +             -
     +  3158            0            3158
Pred
     -    82           6760            6842

          3240          6760           10000

Accuracy = 0.9918
```

Figure 6: Result on training set

```
:- modeh(*, illegal(+pos, +pos, + pos, + pos, + pos, +pos)).

% encoding of illegal positions
:- modeb(*, adj(+pos, +pos)).
:- modeb(*, lt(+pos, +pos)).
:- modeb(*, not(lt(+pos, +pos))).


% determinations for illegal/6
:- determination(illegal/6, adj/2).
:- determination(illegal/6, lt/2).
:- determination(illegal/6, not/1). % add not in order to be able to use it in rules


:- set(clauselength, 6).
:- set(minpos, 70).  % only consider rules that cover at least 70 examples
```

Figure 7: Encoding Aleph simplified version

2. `illegal(A,B,C,D,E,F) :- adj(A,E), adj(B,F).` It is indeed illegal that the BK and the WK are adjacent.

3. `illegal(A,B,C,D,E,F) :- lt(E,A), same(E,C).` This is an illegal position since the BK is in check by the WR.

Test the theory on the provided test set One can test how many examples are covered by the learned theory by using `test("train.f",noshow,Covered,Total).` and `test("train.n",noshow,Covered,Total` for the negative examples. With out rule set we could cover 3284 from a total of 3361 positive examples from the test set and covered 0 from 6639 negative examples. Therefore we have misclassified 0,77% examples of the test set.

Simpler encoding less accuracy We had quite a lot of mode declaration and additional predicates in order to learn a rule set with Aleph. The up until now described version achieved an accuracy of 99.18% on the training set and only misclassified 0.77% on the test set with a rule set consisting of 7 rules. However, after looking closer into the encoding for Popper, we were able to simplify the encoding of Aleph. The simplified encoding looks as shown in figure 7 . This encoding learns 9 rules and achieves the result shown in figure 8. The simplified version performs slightly worse on the test set.

Learn a rule set with Popper To be able to learn a rule set with Popper we have to encode the Aleph file into three Popper files. `bias.pl`, `bk.pl` consisting of the background knowledge and

```
[Training set performance]
            Actual
          +             -
     +  3163            0            3163
Pred
     -    77           6760            6837

          3240          6760           10000

Accuracy = 0.9923
[Training set summary] [[3163,0,77,6760]]
[time taken] [13.666880999999998]
[total clauses constructed] [26781]
true.

?- test("test.f",noshow,Covered,Total).
Covered = 3293,
Total = 3361.

?- test("test.n",noshow,Covered,Total).
Covered = 2,
Total = 6639.
```

Figure 8: Result Aleph simplified version

```
max_vars(6). % max number of distinct variables
max_body(4). % max number of body predicates
max_clauses(4). % max number of clauses

head_pred(illegal, 6). % target predicate

% body predicates
body_pred(adj, 2).
body_pred(lt, 2). % less than
body_pred(gt, 2). % greater than - not(lt/2))
body_pred(same, 2). % same position
body_pred(sameb, 3). % same row or column

% number of arguments each predicate takes
type(illegal, (pos, pos, pos, pos, pos, pos)).
type(adj, (pos, pos)).
type(lt, (pos, pos)).
type(gt, (pos, pos)).
type(same, (pos, pos)).
type(sameb, (pos, pos, pos)).

% all arguments are inputs
direction(illegal, (in, in, in, in, in, in)).
direction(adj, (in, in)).
direction(lt, (in, in)).
direction(gt, (in, in)).
direction(same, (in, in)).
direction(sameb, (in, in, in)).
```

Figure 9: Encoding Popper

```
********** SOLUTION **********
Precision:1.00 Recall:1.00 TP:3240 FN:0 TN:3240 FP:0 Size:118
illegal(V0,V1,V2,V3,V4,V5):- same(V0,V2),gt(V5,V4),same(V3,V1).
illegal(V0,V1,V2,V3,V4,V5):- same(V3,V1),same(V0,V2),lt(V5,V4).
illegal(V0,V1,V2,V3,V4,V5):- gt(V1,V3),lt(V5,V0),same(V2,V4).
illegal(V0,V1,V2,V3,V4,V5):- sameb(V0,V4,V2),lt(V1,V3),gt(V5,V1).
illegal(V0,V1,V2,V3,V4,V5):- sameb(V2,V4,V5),lt(V4,V0),lt(V1,V3).
illegal(V0,V1,V2,V3,V4,V5):- adj(V0,V4),adj(V5,V1),gt(V2,V3).
illegal(V0,V1,V2,V3,V4,V5):- lt(V0,V1),same(V3,V5),lt(V2,V4).
illegal(V0,V1,V2,V3,V4,V5):- adj(V0,V4),adj(V5,V1),gt(V3,V2).
illegal(V0,V1,V2,V3,V4,V5):- lt(V1,V2),gt(V4,V0),same(V3,V5).
illegal(V0,V1,V2,V3,V4,V5):- adj(V5,V1),same(V4,V2),lt(V0,V3).
illegal(V0,V1,V2,V3,V4,V5):- lt(V0,V1),same(V3,V5),gt(V2,V4).
illegal(V0,V1,V2,V3,V4,V5):- sameb(V5,V3,V1),gt(V0,V2),gt(V0,V4).
illegal(V0,V1,V2,V3,V4,V5):- same(V5,V3),lt(V3,V1),gt(V4,V2),gt(V0,V1).
illegal(V0,V1,V2,V3,V4,V5):- lt(V0,V4),lt(V0,V1),same(V2,V4),gt(V3,V5).
illegal(V0,V1,V2,V3,V4,V5):- same(V5,V3),gt(V4,V2),lt(V1,V3),gt(V0,V1).
illegal(V0,V1,V2,V3,V4,V5):- lt(V0,V4),lt(V0,V1),same(V2,V4),lt(V3,V5).
illegal(V0,V1,V2,V3,V4,V5):- lt(V0,V4),same(V4,V2),gt(V3,V5),gt(V0,V1).
illegal(V0,V1,V2,V3,V4,V5):- lt(V5,V3),same(V4,V2),lt(V4,V0),gt(V0,V1).
illegal(V0,V1,V2,V3,V4,V5):- lt(V0,V1),same(V4,V2),lt(V4,V0),lt(V3,V5).
illegal(V0,V1,V2,V3,V4,V5):- lt(V0,V4),gt(V5,V3),same(V2,V4),gt(V0,V1).
illegal(V0,V1,V2,V3,V4,V5):- same(V5,V3),lt(V1,V3),gt(V2,V4),gt(V0,V1).
illegal(V0,V1,V2,V3,V4,V5):- same(V5,V3),lt(V3,V1),lt(V4,V2),gt(V0,V1).
illegal(V0,V1,V2,V3,V4,V5):- lt(V0,V1),same(V4,V2),lt(V5,V3),adj(V1,V0).
illegal(V0,V1,V2,V3,V4,V5):- lt(V0,V1),same(V4,V2),gt(V1,V3),gt(V1,V5).
illegal(V0,V1,V2,V3,V4,V5):- same(V5,V0),lt(V0,V1),same(V4,V2),adj(V1,V3).
illegal(V0,V1,V2,V3,V4,V5):- gt(V5,V3),same(V4,V2),lt(V4,V0),gt(V0,V1).
******************************
Num. programs: 474512
```

Figure 10: Rule set Popper

`exs.pl` encoding the positive and negative examples. For details on the encoding we refer directly to the files.

Popper generated `V0` to `V5` to encode the variables `colwk, rowwk, colwr, rowwr, colbk, rowbk` within the predicate definition. The predicate `gt` refers to "greater than" representing `not(lt/2))`.

Popper learned the rule set shown in figure 10 with a recall of 1 indicating that 100% of the actual illegal positions were correctly identified by the learned rules.

Comparison between Aleph and Popper Both system achieve a high precision and recall. Popper generates a larger number of rules (26), which indicates a more detailed approach to capture illegal positions. However a major drawback is the long running time (20 minutes) to obtain a rule set. In comparison Aleph only generates 7 rules and is much faster than Popper. The rules of Aleph are more general, making them easier to interpret.

Exercise 4

In bonus exercise, we employed Aleph, to engage in an elaborate experiment aimed at deducing logical rules from a given dataset (from assignment 1). The primary objective was to determine sequences of arithmetic operations that could transform a list of integers into a specified target number. We prepared a diverse set of examples, incorporating a variety of arithmetic operations

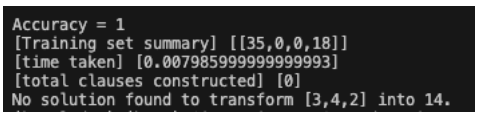

Figure 11: Results of Aleph: exercise 4

such as addition, subtraction, multiplication, and division, applied across arrays of differing lengths. These examples were carefully designed to cover a broad spectrum of possible arithmetic transformations, providing the model with both straightforward and complex computational challenges.

Despite the extensive dataset and the robust capabilities of the Aleph system, the model encountered significant challenges in generalizing the learned rules to apply to new, unseen scenarios. Throughout the exercise, it became apparent that while Aleph could effectively learn from the examples provided, its ability to extrapolate and apply these learnings to solve novel problems without explicit prior examples was limited. The exploration provided valuable insights into the constraints of current inductive logic programming tools when faced with complex logical reasoning tasks in arithmetic contexts.

Conclusion

In this assignment, we explored the capabilities of two ILP systems, Aleph and Popper, across different learning tasks. Both systems demonstrated high accuracy in learning rules from examples, with Aleph excelling in speed and simplicity, while Popper offered more detailed and complex rule generation. The comparison highlighted that while Aleph is suitable for tasks requiring quick hypothesis generation, Popper is advantageous for more complex problems requiring deeper rule induction. The experiments provided valuable insights into the strengths and limitations of each system, underscoring the importance of selecting the appropriate tool based on the specific requirements of the learning task.