Solving the Maze Problem with Inductive Logic Programming: A comparison between HYPER, Metagol and ILASP

Angelo Andreussi, Alex Della Schiava, Claudia Maußner

July 5, 2021

1 Outline

In this document, we intend to describe our Inductive Logic Programming (ILP) solutions to the Maze problem.
Section 2 offers a brief illustration of the Maze Problem we have been working on, including the main choices and assumptions we made.
Section 3 lists the tools we have used to reach our goal with a brief description on how they work.
Section 4 thoroughly illustrates our main implementations for the learning tasks. Finally **??** provides our final conclusions on our work, supported by the data gathered from our implementations.

2 Introduction

The main objective of this project is to use different Inductive Logic Programming (ILP) techniques on the same problem in order to highlight their differences. Despite the importance of performance differences (see Section 5), we are also going to focus on the differences concerning the approach to the problem, as some of us had to take

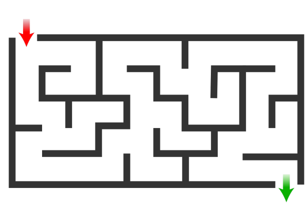

Figure 1: Example of a Maze

completely different paths in order to reach similar goals.

Our work is focussed on the Maze problem. This problem consists in finding a path from point `A` to point `B` in a labirinth-like shaped map (see Figure 1). A variety of classical algorithms can be used to solve this problem, starting from the most naïve wall following algorithm to more complex and elaborated ones exploiting graph theory concepts.
By approaching this simple problem with ILP though, it is possible to extend it into a much more sophisticated and interesting problem. For instance, it allowed us to start with the assumption that the problem's main character (the one we shall refer to as *agent*) has no knowledge about *how* to move. Consequently, before even trying to solve the Maze, the agent needs to *learn* what a *move* is and, more specifically, what a *legal* move is. The second step consisted into *teaching* the agent how to reach two distant cells. Lastly, in order to solve the Maze, it is either possible to keep using ILP in order to find a solution or use the learned rules in order to implement them in a logic programming model of a planning problem.

Getting more specific on our problem's instance, we decided to use a $n * m$ grid as a map. The Maze is defined by placing an arbitrary number of obstacles on the grid. A cell in the grid is identified by a pair `(X,Y)`, where `X` and `Y` are coordinates which use the top-left corner as origin.

3 Background

3.1 HYPER

3.2 Metagol

Metagol is a system used for ILP which relies on meta-interpretative learning.
Using Metagol, four key components need to be defined:

- **Metarules** (M)**.** Metarules are used to define the *language bias* of the task. A large number metarules allows for a less strict language bias, hence a larger search space in which to find a solution.
- **Background Knowledge** (BK)**.** The knowledge the system is initially assumed to have about the task to be carried out. It is a set of Prolog rules that the system can use either directly or indirectly in order to induce the hypothesis.
- **Positive Examples** (E^+)**.**
- **Negative Examples** (E^-)**.**

With these four components defined, Metagol will try to find a solution running the following algorithm:

1. Select a positive example to be proven.

2. Try to prove the example using the existing BK or previously induced clauses.

3. (If step 2 did not work) Unify the example with the head of a metarule and repeat steps 1,2 and 3 for each atom in the body of the obtained rule.

4. Once the hypothesis is proven to be complete (all the positive examples have been proven and covered), test its consistency. If any negative example is covered, backtrack to a choice made in step 3 which, supposedly, led to this situation.

In this brief illustration of Metagol, the process of *predicate invention* is not covered due to a lack of time to further study it. Our findings about this process mainly derive from experimental experience and have no theoretical backup. Nonetheless, we will still point out the influence it had on our results.

3.3 ILASP

ILASP enables learning programs containing normal rules, choice rules and both hard and weak constraints, these are the rules that compose ASP econdings and here this tool will be used for the maze problem. Weak constraints won't be covered, the goal here is to try to learn some rules that will form an encoding of that problem.

Similarly to what presented above for Metagol, providing to ILASP Background knowledge, language Bias, Positive and negative examples, it's possible to learn rules inductivly.

4 Implementation

4.1 HYPER

4.2 Metagol

The scripts discussed in this section of the report can be found in the `Metagol` folder.

4.2.1 `learning_to_walk.pl`

Since learning the predicate `adjacent/2` was quite trivial, the first task to be learned consists moving from one cell to another adjacent, legal one. Differently from the other two implementations, here the `adjacent/2` predicate is learned indirectly through *predicate invention*.

In this case, the system is assumed to know what a legal cell is and, given a pair of coordinates, how to increment/decrease the value of a single coordinate.

```
body_pred(inc_x/2).
body_pred(dec_x/2).
body_pred(inc_y/2).
body_pred(dec_y/2).
body_pred(legal_position/1).
```

Listing 1: Background Knowledge to walk

The positive and negative examples used in this program are quite straightforward to illustrate. A positive example is needed for each possible direction (up, down, left, right). Having the predicate `legal_position/1` as part of the background knowledge further simplifies the task of defining the negative examples, as we only need four of them: one illegal move for each direction. This is possible because `legal_position/1` is false whatever kind of illegal position it is considering (out of bounds or obstacle).

The spotlight of this task is on metarules. Metarules define the search space and, therefore, they also have a huge impact on both performance and the way the solution is presented. In this case the metarules used are the following:

```
metarule(ident, [P,Q], [P,A,B], [[Q,A,B]]). % Identity
metarule(postcon, [P,Q,R], [P,A,B], [[Q,A,B], [R,B]]). % Postcondition
metarule(i_postcon, [P,Q,R], [P,A,B], [[R,B], [Q,A,B]]). % Inverted postcondition
```

Listing 2: Metarules in `learning_to_walk.pl`

Focussing on `postcon` and `i_postcon`, it is possible to notice how they are basically defining the same clause shape, just with two inverted atoms in the body. The results, though, will show how much difference using one metarule or another can make.

```
move(A,B):-inc_x(A,B),legal_position(B).
move(A,B):-inc_y(A,B),legal_position(B).
move(A,B):-dec_x(A,B),legal_position(B).
move(A,B):-dec_y(A,B),legal_position(B).
```

Listing 3: Result of `postcon`

```
move(A,B):-legal_position(B),move_1(A,B).
move_1(A,B):-inc_x(A,B).
move_1(A,B):-inc_y(A,B).
move_1(A,B):-dec_x(A,B).
move_1(A,B):-dec_y(A,B).
```

Listing 4: Result of `i_postcon`

As noticeable in Listing 3, the solution is more succinct, with only 4 clauses used for the solution. Using `i_postcon` however, allows to learn the predicate `move_1` which corresponds to the predicate `adjacent/2` previously mentioned.

Before diving into the timing analysis for these two cases, a third one deserves to be mentioned where `legal_position/1` is replaced by the two conjuncts that defined it, namely `is_free/1` (which checks whether a position is not an obstacle) and `in_range/1` (which checks if a position is in bounds). In order to work on this task, a new metarule

is introduced: `metarule([[P,Q,R,S],[P,A,B],[[Q,A,B],[R,B],[S,B]]).])`. This metarule, to which we will refer to as *double postcondition* (`double_postcon`) shapes the resulting clause with two different postconditions.
The result is quite intuitive:

```
move(A,B):-inc_x(A,B),in_range(B),is_free(B).
move(A,B):-inc_y(A,B),in_range(B),is_free(B).
move(A,B):-dec_x(A,B),in_range(B),in_range(B).
move(A,B):-dec_y(A,B),in_range(B),is_free(B).
```

Listing 5: Result of `double_postcon` result

Finally, Table 1 offers an overview of the timings of these three slightly different implementations. These results highlight the impact that one more clause or just a slightly increased clause length can have on time performance. Concluding, Metagol is affected by a trade-off between expressiveness and performance.

`postcon`	`i_postcon`	`double_postcon`
0.047	0.121	0.156

Table 1: `learning_to_walk.pl` performances (seconds).

4.2.2 `learning_to_travel_with_memory.pl`

This program's task is to learn the predicate `reach(A,B,L)`, where `A` and `B` are cells and `L` is a list containing the path from `A` to `B`.
In this case, the background knowledge already includes the predicate `move/2` previously learned with `learning_to_walk.pl`. The metarules used are the following:

```
metarule(recursion, [P,Q], [P,A,B,[A|L1]], [[Q,A,C], [P,C,B,L1]]).
metarule(ident, [P,Q], [P,A,B], [[Q,A,B]]).
metarule(ident2, [P,Q], [P,A,B,[A,B]], [[Q,A,B]]).
```

Listing 6: Metarules in `learning_to_travel_with_memory.pl`

Focussing on the `recursion` metarule in Listing 6, its recursive component lays in the fact that it enforces to reuse in its body the predicate used in the head (`P`). It is fair to mention that no elaborated technique was used in order to understand which metarule would fit best for the given task. This metarule was simply chosen because it represented the rule shape we would have used to *"manually"* define `reach/3`.

One interesting thing about this implementation is about the number of examples being used: one positive example and no negative ones. This is because of how small the search space is. Having the background knowledge only including `move/2` means already getting rid of any hypothesis of the predicate `reach/3` that would allow illegal moves. To phrase it in a simpler way, an *agent* that only knows how to move legally

will not be able to make sequences of moves including illegal ones.
Plus, not only is the search space quite small, the *language bias* is also very restrictive: consider Metagol trying to prove the example `reach((1,1), (3,1), [(1,1), (2,1), (3,1)])`. The only valid metarule to which unify this example is `recursion`.
Follows the result obtained from the implementation:

```
reach(A,B,[A,B]):-move(A,B).
reach(A,B,[A|C]):-move(A,D),reach(D,B,C).
```

Listing 7: Result of `learning_to_travel_with_memory.pl`

4.2.3 `reach_from_scratch_memory.pl`

This program's task is analogous to the one of the program illustrated at Section 4.2.2. This time, though, the background knowledge is the same as for `learning_to_walk.pl`, shown in Listing 1.
As a result of this, the system will have to deal with a large search space, this means that, at last, the spotlights points onto the examples.
Given that the system now does not initially know how to `move/2`, the positive examples will have to include a successful *walk* from a legal cell to another adjacent one for each of the four directions. Last, only an example of one successful path between two distant cells is needed. It is important for this path to contain moves in all four directions. The negative examples are the same as it was for the first implementation described in Section 4.2.2: one illegal move for each direction. The examples can be visualized more clearly in Listing 8.

```
Pos = [
    reach((1,1), (2,1), [(1,1), (2,1)]),
    reach((4,4), (4,3), [(4,4), (4,3)]),
    reach((5,2), (5,3), [(5,2), (5,3)]),
    reach((5,5), (4,5), [(5,5), (4,5)]),
    reach((3,1), (4,5), [(3,1), (4,1), (4,2), (4,3), (5,3), (5,4), (5,5), (4,5)])
],
Neg = [
    reach((5,1), (6,1), [(5,1), (6,1)]),
    reach((1,1), (1,0), [(1,1), (1,0)]),
    reach((5,5), (5,6), [(5,5), (5,6)]),
    reach((1,1), (0,1), [(1,1), (0,1)])
]
```

Listing 8: Examples in `reach_from_scratch_memory.pl`

Surprisingly enough, no examples of paths containing incoherent moves were needed. But as previously mentioned, this should be justifiable by the restricting language bias. Following, the metarules (Listing 9) and the result of the implementation (Listing 10).

```
metarule(ident, [P,Q], [P,A,B], [[Q,A,B]]).
metarule(ident2, [P,Q], [P,A,B,[A,B]], [[Q,A,B]]).
metarule(postcon, [P,Q,R], [P,A,B], [[R,B], [Q,A,B]]).
metarule(recursion, [P,Q,R], [P,A,B,[A|L1]], [[R,B], [Q,A,C], [R,C], [P,C,B,L1]]).
```

Listing 9: Metarules in `reach_from_scratch_memory.pl`

```
reach(A,B,[A,B]):-reach_1(A,B).
reach_1(A,B):-legal_position(B),reach_2(A,B).
reach_2(A,B):-inc_x(A,B).
reach_2(A,B):-dec_y(A,B).
reach_2(A,B):-inc_y(A,B).
reach_2(A,B):-dec_x(A,B).
reach(A,B,[A|C]):-legal_position(B),reach_2(A,D),legal_position(D),reach(D,B,C).
```

Listing 10: Result of `reach_from_scratch_memory.pl`

On a last note, the metarule `recursion` in this case needed to be added a post-condition (on cell `B`) and a *"middle-condition"* (on cell `C`). It was quite hard to figure out why, but removing one of these atoms would lead to an endless execution of the program at a number of clauses equal to 2.

4.2.4 Other tasks

- `learn_to_win.pl`. Analogously to Cropper's example `robot.pl`, this program is able to offer the solution of the given Maze problem.
- `tail_lttm.pl` (not working). As it was for the implementation described in Section 4.2.2, this program is also used to learn the predicate `reach/3`, this time with *tail recursion*. The reason why this work was deemed relevant is offered at Section 5.

4.3 ILASP

4.3.1 Learning normal rules - learning how to walk

The first task is learning to walk on the maze, considering adjacent cells and the obstacles (walls and co.). This task have been split for complexity reasons, as a result first it will be learned how to move on near cells and then obstacles will be considered, in 2 different ILASP scripts. Here some normal rules will be learned, other kind of rules have been learned on other scripts but regarding another type of ASP model.

4.3.2 Learning to walk on adjacent cells

This is the ilasp code written for the pourpose, with some bit of background knowledge, definition of search space with language bias and some examples with all the different "cases". Finding those examples have been pretty difficult because they must be "meaningful" and as such must capture all the different contexts and casistics.

```
%%%%%%%%%%%%%%%%%%%%%%%%%learn how to move on near cells
row(1..5).
col(1..5).

cell(X,Y) :- row(X), col(Y).

succ(0,1).
succ(X, X+1) :- cell(X,_).

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%SEARCH_SPACE + EXAMPLES
#pos(p1, {next((4,2), (4,1)), next((4,2), (4,3)), next((4,2), (3,2)), next((4,2),
    (5,2))}, {}).
#pos(p2, {next((2,3), (2,2)), next((2,3), (1,3)), next((2,3), (2,4)), next((2,3),
    (3,3))}, {}).

%no out of range or jump
#neg(a, {next((1,0), (1,1))}, {}).
#neg(b, {next((1,1), (0,1))}, {}).
#neg(c, {next((0,1), (1,1))}, {}).
#neg(d, {next((1,1), (1,0))}, {}).
#neg(e, {next((5,5), (6,5))}, {}).
#neg(f, {next((5,5), (5,6))}, {}).
#neg(g, {next((6,5), (5,5))}, {}).
#neg(h, {next((5,6), (5,5))}, {}).
%no diagonal move
#neg(i, {next((2,4), (1,3))}, {}).
#neg(l, {next((2,4), (1,5))}, {}).
#neg(m, {next((2,4), (3,5))}, {}).
#neg(n, {next((2,4), (3,3))}, {}).
%no move same cell
#neg(o, {next((2,4), (2,4))}, {}).

#modeb(2, cell(var(r), var(c)), (positive, anti_reflexive)).
#modeb(1, succ(var(c), var(c)), (positive, anti_reflexive)).
#modeb(1, succ(var(r), var(r)), (positive, anti_reflexive)).
#modeh(next((var(r), var(c)), (var(r), var(c)))).

#maxv(3).
```

Successor predicate have been inserted, it rapresent the simple "arithmetic" concept of successive numbers, it's important to define because ILASP doesn't include this "sum" operator bilt-in. In the language bias definition I used the "positive" and "anti-reflexive" options to reduce the search-space and found earlier the result. This options could be avoided. At 2 the output of the script, with the learned rules, ILASP actually learned that "next" predicate is true for adjacent cells, based on that the movement on the grid will be possible. NB: this is equivalent to the predicate "adjacent" learned by my companions

```
agnul@agnul-H87-HD3:~/gitHub/ATAI_Maze_Project/ILASP/ILASP_TASKS/commonStuff$ ILASP4 --version=3 learnMove.las
next((V1,V2),(V3,V2)) :- cell(V1,V2); cell(V3,V2); succ(V3,V1).
next((V1,V2),(V3,V2)) :- cell(V1,V2); cell(V3,V2); succ(V1,V3).
next((V1,V2),(V1,V3)) :- cell(V1,V2); cell(V1,V3); succ(V3,V2).
next((V1,V2),(V1,V3)) :- cell(V1,V2); cell(V1,V3); succ(V2,V3).

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%% Pre-processing                          : 0.002s
%% Hypothesis Space Generation             : 0.337s
%% Conflict analysis                       : 3.146s
%%    - Negative Examples                  : 0.276s
%%    - Positive Examples                  : 2.87s
%% Counterexample search                   : 0.154s
%%    - CDOEs                              : 0.003s
%%    - CDPIs                              : 0.151s
%% Hypothesis Search                       : 0.285s
%% Propagation                             : 0.804s
%%    - CDPIs                              : 0.804s
%% Total                                   : 4.776s
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
```

Figure 2: Learned rules - adjacent move

4.3.3 Learning to walk on cells without obstacles

The next step is to consider obstacles on the grid, as discussed in the Introduction part of the report. Here the goal is to find a new normal rule that represent the concet of a "valid move", in a cell without obstacles. Initially, a maze is defined, this is the same maze used by my companions for our work. This is the task written for the purpose:

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%learn how to move on cells without obstacles
row(1..5).
col(1..5).

obstacle(1,2).
obstacle(2,2).
obstacle(3,2).
obstacle(3,3).
obstacle(4,3).
obstacle(4,4).
obstacle(3,4).
obstacle(2,4).
obstacle(1,4).
obstacle(1,5).
obstacle(5,1).

start(1,1).
goal(5,5).

%%%%%%%%%%%%%%

cell(X,Y) :- row(X), col(Y).

succ(0,1).
succ(X, X+1) :- cell(X,_).

%PATHS ADJACENTS (learned from previous ilasp task)
next((V1,V2),(V3,V2)) :- cell(V1,V2), cell(V3,V2), succ(V3,V1).
next((V1,V2),(V3,V2)) :- cell(V1,V2), cell(V3,V2), succ(V1,V3).
next((V1,V2),(V1,V3)) :- cell(V1,V2), cell(V1,V3), succ(V3,V2).
next((V1,V2),(V1,V3)) :- cell(V1,V2), cell(V1,V3), succ(V2,V3).


%%%%%%%%%%%%%%%%%%%%%%%%%%%%%SEARCH_SPACE + EXAMPLES

#pos(po, {nextLegit((1,1),(2,1))}, {}).
#pos(po2, {nextLegit((4,1),(4,2))}, {}).

%no movement on obstacles
#neg(a, {nextLegit((1,1),(1,2))}, {}).
#neg(av, {nextLegit((3,2),(4,2))}, {}).
#neg(af, {nextLegit((1,2),(1,1))}, {}).
#neg(b, {nextLegit((4,1),(5,1))}, {}).
#neg(g, {nextLegit((3,3),(2,3))}, {}).

#modeb(1, next((var(r), var(c)), (var(r), var(c)))).
#modeb(2, obstacle(var(r), var(c))).
#modeh(1, nextLegit((var(r), var(c)), (var(r), var(c)))).

#maxv(3).
```

In the code the rules learned previously have been inserted and used in the search space. Negative examples shows that no movement is possible TO obstacles and FROM obstacles, this 2 casistics are important for a correct learning. At 3 the output of the script, with the learned rules, ILASP actually learned this new "nextLegit" predicate that represent the concept of a valid move: a move from (or to) cells without obstacles.

NB: this is equivalent to the "move" predicate learned by my companions.

```
agnul@agnul-H87-HD3:~/gitHub/ATAI_Maze_Project/ILASP/ILASP_TASKS/commonStuff$ ILASP4 --version=3 learnNoObstacles_simple.las
nextLegit((V1,V2),(V3,V2)) :- not obstacle(V1,V2); not obstacle(V3,V2); next((V3,V2),(V1,V2)).
nextLegit((V1,V2),(V1,V3)) :- not obstacle(V1,V2); not obstacle(V1,V3); next((V1,V2),(V1,V3)).

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%% Pre-processing                          : 0.002s
%% Hypothesis Space Generation             : 2.191s
%% Conflict analysis                       : 2.289s
%%    - Negative Examples                  : 1.54s
%%    - Positive Examples                  : 0.748s
%% Counterexample search                   : 0.085s
%%    - CDOEs                              : 0s
%%    - CDPIs                              : 0.083s
%% Hypothesis Search                       : 0.058s
%% Propagation                             : 0.825s
%%    - CDPIs                              : 0.825s
%% Total                                   : 5.491s
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
```

Figure 3: Learned rules - avoid obstacles

It's interesting to see that, setting 3 variables in the language bias, ilasp learned just 2 rules: it considers the cases when near cells share the same horizontal or vertical direction. Probably, I would have write 4 rules using 4 different variables in a naive way, considering the "4 directions adjacency", but this tool finds a more compact solution.

4.3.4 using this rules to define an ASP model

having this rules in hand now is possible to define a model that solves our problem. Learned rules are reported exactly as they were on the ILASP scripts output. Some pieces are missing: some other rules need to be defined but they are quite intiuitive at this point. The maze reported on the encoding is the same used for the ilasp task.

```
%MODEL THAT SOLVES PROBLEM OF PATHFINDING IN THE GRID
row(1..5).
col(1..5).

obstacle(1,2).
obstacle(2,2).
obstacle(3,2).
obstacle(4,4).
obstacle(3,4).
obstacle(2,4).
obstacle(1,4).
obstacle(1,5).
obstacle(5,1).

start(1,1).
goal(2,5).

%for each position define cell pred.
cell(X,Y) :- row(X), col(Y).

%FIND A PATH FROM START
move(0,0,X,Y) :- start(X,Y).
%for each "move" find another linked to it that is a legit move!
1{move(X,Y,X1,Y1): nextLegit((X,Y), (X1,Y1))}1:- move(_,_, X,Y), not goal(X,Y).

succ(0,1).
succ(X, X+1) :- cell(X,_).

%LEARNED BY ILASP, move on adj cells.
next((V1,V2),(V3,V2)) :- cell(V1,V2), cell(V3,V2), succ(V3,V1).
next((V1,V2),(V3,V2)) :- cell(V1,V2), cell(V3,V2), succ(V1,V3).
next((V1,V2),(V1,V3)) :- cell(V1,V2), cell(V1,V3), succ(V3,V2).
next((V1,V2),(V1,V3)) :- cell(V1,V2), cell(V1,V3), succ(V2,V3).

%LEARNED BY ILASP, move on adj cells. without obstacles
nextLegit((V1,V2),(V3,V2)) :- not obstacle(V1,V2); not obstacle(V3,V2); next((V3,V2)
    ,(V1,V2)).
nextLegit((V1,V2),(V1,V3)) :- not obstacle(V1,V2); not obstacle(V1,V3); next((V1,V2)
    ,(V1,V3)).

%ON GOAL POSITION STOP
:- goal(X,Y), not move(_,_, X,Y).

#show move/4.
```

at 4 the execution of "clingo" command on this model is shown: the path is represented by the chaning "move" predicate.

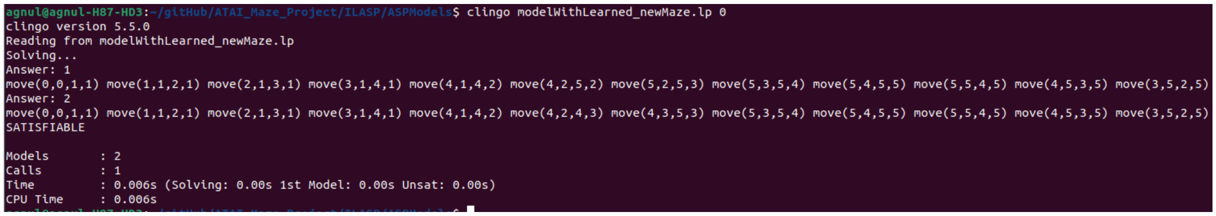

Figure 4: Path found on the grid

4.3.5 performance test - scalability

The work done with ilasp shows clearly this fact: the time complexity doesn't scale well with respect to the search space dimension. In fact, when trying to learn the "move to adjacent cells AND without obstacles" (the 2 tasks on the same scrit) compexity costs exploded "simply" for the insertion of the predicate "obstacle". (for a total of 6 predicates in the search space). it is quite evident that a sort of exponential trend is in place and here I will like to do a specific test to demonstrate this.

Using the "learn to walk on adjacent cells" ilasp script I tried to augment the search space and see time needed for computation, the augmenting has been done insertig other predicates in language bias, modyfing language bias to insert more "usage" of the same predicate, eliminating the "positive" and "anti-reflexing" options on language bias. After that, an analysis on computation time vs search space dimension (measured as the size of rules in the search space) has been conducted, at 5 the graphical results, the blue points are the instantes of the benchmarks. Interestingly other tests conducted with a searh space grater than 200 lead to huge times like 2 hours.

The grow shows an evident simil-exponential trend, especially the steep from the "57 seconds" point to the "200" one.

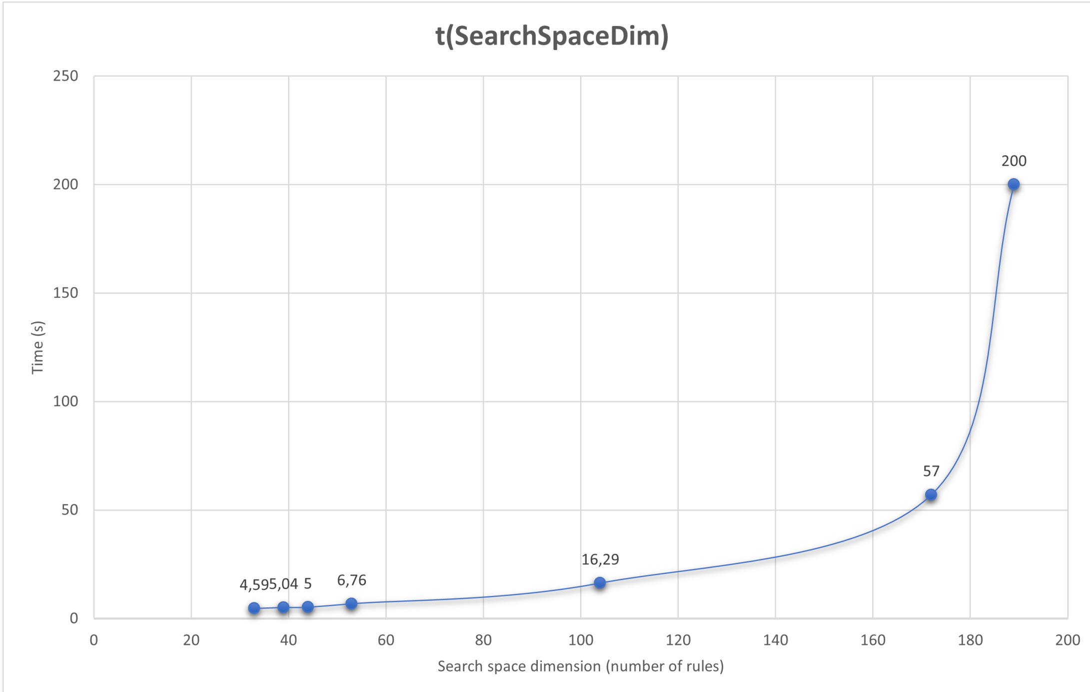

Figure 5: performance test result

5 Conclusions

In this section we will offer some conclusions on our works, mainly by comparing them. We deemed important to compare HYPER, Metagol and ILASP from both the points of view of the performances and the approaches used to tackle the Maze problem.

5.1 Performance Analysis

Table 2 offers an overview of the timings of the main tasks presented in Section 4.

Task	**HYPER**	**Metagol**	**ILASP**
`adjacent/2`	175.884	0.056	4.767
`move/2`	0.063	0.047	5.343
`move/2` ($7 * 7$ grid)	INSERT	0.054	5.432
`reach/3`	1.577	0.027	NA
`move/2` and `reach/3`	16.585	0.848	NA

Table 2: Time comparison between the different systems for the main tasks (seconds)

These timings allow us to conclude that the two grid dimensions used ($5 * 5$ and $7 * 7$) had no impact on the task being learned.
The three systems also share the same behavior when put in front of *combined learning*. In fact, all three systems have shown to perform far better when learning one task at a time than when learning more of them together (`move/2` and `reach/3`).
Table 2 also shows a strong difference in performances between HYPER and Metagol. Given their similar approach, one could expect to also have similar performances. While HYPER uses a more general approach, Metagol owes its efficiency to metarules and the way they shape the language bias. This, though, does not come for free, since defining good metarules requires a very accurate initial idea of what the final solution should be like.

Lastly, Table 3 shows the differences in execution time when using more positive examples than the minimum required. Metagol shows a slight increase in computation time. This could be explained by the fact that Metagol needs to prove more positive examples, even though it has already found the right hypothesis.

$\|E^+\|$	**HYPER**	**Metagol**	**ILASP**
4	INSERT	0.056	5.31
8	INSERT	0.068	5.43
12	INSERT	0.072	5.53

Table 3: Time comparison with increasing positive exmaples (seconds)

5.2 Learning `reach/3` with *tail recursion*

Being *"solving the Maze problem"* part of the title and one of the main goals of this project, we were quite surprised that the `reach/3` predicate learned in our implementations was not able to find a path in our Maze (Figure 6).
By studying the trace when querying Prolog with `reach((1,1), (2,5), L)`, we noticed that the search of the path would get stuck into a loop, going back and forth from cells `(5,2)` and `(5,3)`. The reason of this behavior is related to the (partly) declarative nature of Prolog. The `reach/3` predicate defined as in Listing 10 falls into a loop because, when getting at cell `(5,2)`, the predicate `reach_2/2` is unified with the head of the first rule found in the program. Since there is no `B` such that `inc_x((5,2),B)`, Metagol will go for `dec_y((5,2), B)`. This unification will not work either since there is no `B` such that `dec_y((5,2), B)` (`(5,1)` is an obstacle). At last the unification is done with the rule at Line 5, with the body consisting to `inc_y((5,2), B)` and hence moving to cell `(5,3)`.
Now again, there is no `B` such that `inc_x((5,3), B)`, so Metagol will unify the `reach_2/2`

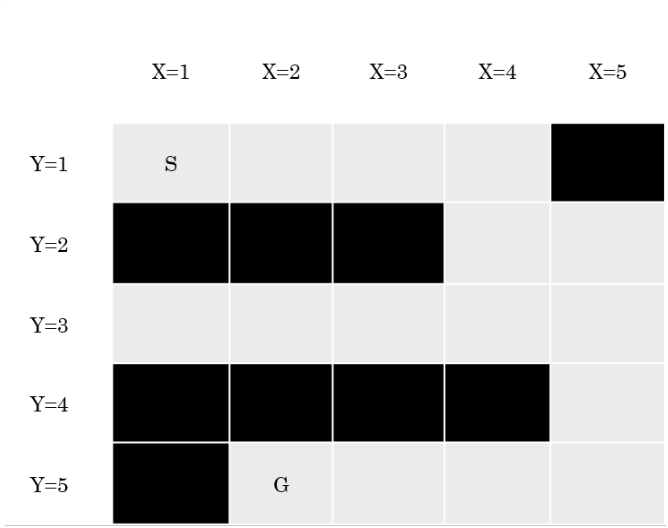

Figure 6: The analyzed Maze

predicate with the head of the rule at Line 4, going for `dec_y((5,3), B)` and, hence, going back onto cell `(5,3)`.

In order to solve this issue we were able to *"manually"* define a procedure for `reach/3` as shown in Listing 11

```
reach(A,B,L) :- reach_1(A,B,[A],L).
reach_1(A,A,L,L).
reach_1(A,B,Acc,L) :-
    move(A,C),
    non_member(C,Acc),
    reach_1(C,B,[C|Acc],L).
```

Listing 11: `reach/3` with tail recursion

This procedure resembles the techniques for a loop preventing Depth-First Search. The idea behind this procedure is to store the already visited cells into an accumulator (`Acc`), and, at each step, check whether a new encountered cell has already been visited before.
Unfortunately, we were not able to learn this procedure through any of the mentioned ILP techniques.