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.

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 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

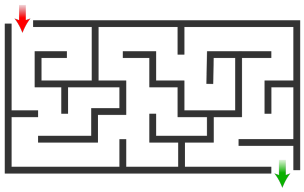

Figure 1: Example of a Maze

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. Follows part of the background knowledge used in order to define the grid:

```
%%% BACKGROUND KNOWLEDGE %%%
width(5).
height(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)).
```

Listing 1: Grid definition

3 Background

4 Implementation

4.1 HYPER

4.2 Metagol

Metagol is a system used for ILP which relies on meta-interpretative learning.
To shortly explain Metagol's learning procedure we will refer to our project, more specifically to the file `Metagol/learn_to_walk.pl`, where the agent learns to move from one cell to an adjacent, available one.
Metagol starts off by trying to prove one of the examples available, in our case we will suppose it would pick the example `move((2,1),(3,1))`. Since the given background knowledge cannot prove this atom and since there are yet no induced clauses, Metagol will try to unify the atom with the head of one of the given metarules, which, in this case, are:

```
1 metarule(ident, [P,Q], [P,A,B], [[Q,A,B]]).
2 metarule(postcon, [P,Q,R], [P,A,B], [[Q,A,B], [R,B]]).
3 metarule(i_postcon, [P,Q,R], [P,A,B], [[R,B], [Q,A,B]]).
```

Listing 2: Metarules in learning_to_walk.pl

We will later elaborate on why we are using the different versions of the postcondition metarule. Supposing Metalog chose `i_postcon` this would result in a substitution of its head with the atom:

4.3 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.

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.

Note: the rules learned here correspond to the predicate "move" learned by my companions.

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 has been pretty difficult because they must be "meaningful" and as such must capture all the different contexts.

```
%%%%%%%%%%%%%%%%%%%%%%%%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).
```

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.

```
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 on the "background knowledge" in (ref). Here the goal is to find a new normal rule that represent the concet of a "valid move", in a cell without obstacles.

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%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).
```

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.

```
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

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 code will be reported with the same grid discussed in ...

```
%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(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).

%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.

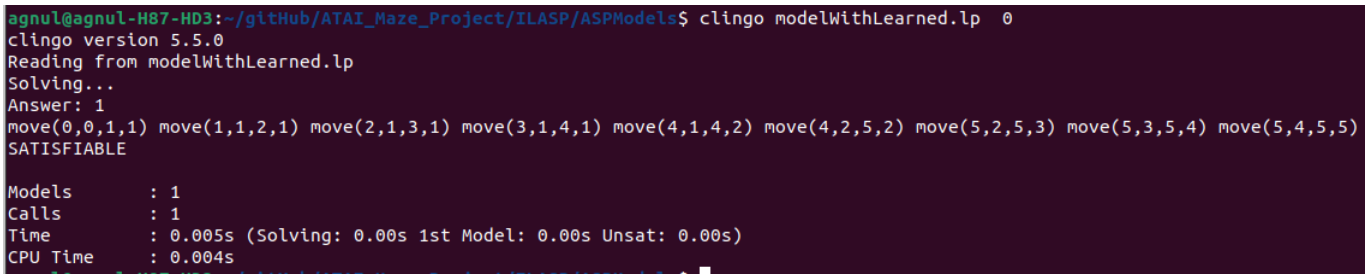

Figure 4: Path found on the grid

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 ¿ 200 lead to

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

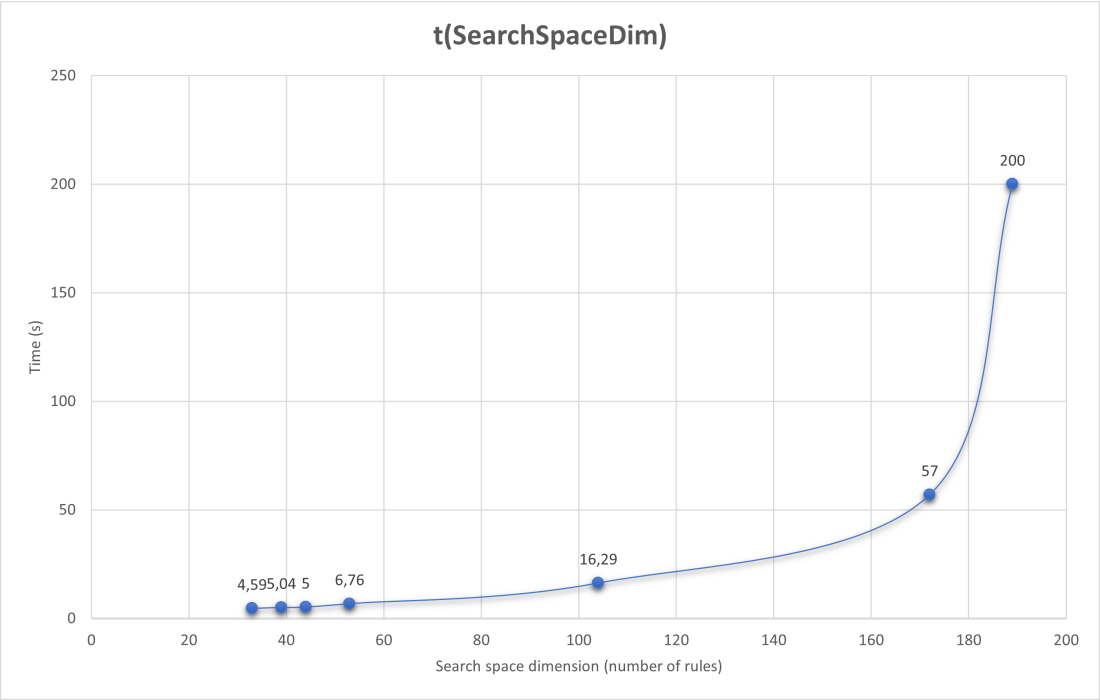

Figure 5: performance test result

5 Performance comparison