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

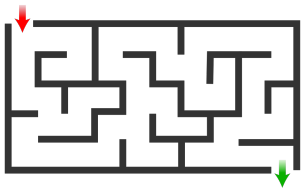

Figure 1: Example of a Maze

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 these metarules, in particular 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 normal, choiche and hard constraints for 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, in the next sections other kind of rules will be covered (regarding the same problem).

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 different "cases". Finding "meaninful" examples have been pretty difficult.

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

Listing 3: Grid definition

5 Performance comparison