Reasoning in High Schools: do it with ASP! *

Agostino Dovier[1], Paolo Benoli[2],
Maria Concetta Brocato[3], Luciano Dereani[3], and Federica Tabacco[3]

[1] Università di Udine, Dipartimento di Scienze Matematiche, Informatiche e Fisiche, CLPLAB. agostino.dovier@uniud.it
[2] ISIS Brignoli-Einaudi-Marconi, Gradisca d'Isonzo–Staranzano
[3] ISIS A. Malignani, Udine

Abstract. We report on a teaching experiment carried on in some high schools of the Friuli Venezia Giulia region. Starting with a two-hours talk on Artificial Intelligence and games proposed two years ago, the project went on with a short course on modeling with Answer Set Programming for students last year. In the current third year a course addressed to (high school) teachers has been organized. The aim is to prepare with them didactical material for their students to be used in the following years in the same schools, and possibly in other schools, without the need of lectures held by an external logic programming expert.

Keywords: Teaching Logic Programming, Modeling, AI and Games.

1 Introduction

The activity we report in this paper was conceived within the so-called *Piano Lauree Scientifiche* (briefly, PLS `http://www.progettolaureescientifiche.eu/`), an Italian (national level) project aimed at spreading some topics not comprised in high school standard curricula with the purpose of attracting students interest towards sciences. After a couple of seminaries on Artificial Intelligence (AI) and the rôle of solving games within AI in the academic year (briefly, A.A. for *Anno Accademico*) 2013/14, a ten-hours course for students on "intuitive" problem solving with Answer Set Programming (ASP) was organized (A.A. 2014/15), and, finally, a twenty-hours course on the same topics was provided to their teachers (A.A. 2015/16). In this last course both intuitive ASP modeling and formal theoretical issues have been presented and discussed. Course participants, together with the course teacher, are preparing lecture notes targeted to high schools students. This way, this material can be spread to high schools without the need of an external "expert" of logic programming.

The historical relevance of (game) problem solving in AI is witnessed by pioneering contributions by Shannon, Zuse, and Turing (e.g. [3]) and by Newell,

* This research is partially supported by INdAM-GNCS 2015 and 2016 projects and by PLS 2010-2014 and 2014-2016 projects.

Shaw, and Simon in the early Fifties on chess playing programs (see, e.g., [10]). Only in 1996 a computer explicitly developed for chess playing (IBM Deep Blue) defeated the world champion Garry Kasparov. In the days this paper was written the deep-learning based computer program AlphaGo was beating the world Go champion Lee Sedol (in January 2016, Fan Gui, the European Go champion, was defeated by AlphaGo, as well). Another witness of the relevance of games for AI is represented by the Angry Birds competition organized in the last issues of ECAI/IJCAI conferences (`https://aibirds.org/`). Moreover, we believe that solving (easier) games and puzzles with AI, and in particular logic programming, techniques can be an appealing way for teenagers to approach problem modeling and, in general, computer programming.

While having fun by solving puzzles students are forced to reason on the "modeling" stage. They are forced to capture the essential, declarative part of the problem. They should use a programming language and therefore learn to avoid syntax errors. They should learn to verify from input/output analysis if their encoding is correct. The use of ASP allows them to use either universal or existential quantification, possibly arbitrarily nested. As it emerged in a discussion at the 2015 GULP meeting in Genova, it is rather common for (even older) students to make bad mistakes when complementing quantifications. ASP modeling can be a way to improve their understanding of these basic, but sometimes not intuitive, notions with positive side effects in their overall education.

2 The course

We focus here on the course for teachers started in Fall 2015. Details and slides used are reported in `www.dimi.uniud.it/dovier/DID/lpandgames.html`. After a first introductory lecture on AI, Knowledge Representation, and games, the syntax and model-theoretical semantics of logic programming were introduced. General and Herbrand Models were presented, as well as the formal and intuitive relationships between minimal/minimum models and logical consequences. The main differences between Prolog and ASP (Turing completeness and naive handling of negative information for the former, limited computational power but excellent modeling capabilities exploiting default negation and stable model semantics for the latter) have been then described. Since the focus of the course was on ASP modeling [11] some parts related to continuity of the T_P and transfinite results (e.g., those concerning with coinductive reasoning and greatest fixed points [2, 1]) were omitted for simplicity. Rather, some efforts were spent on the intutive side of the notions of supported and stable models [9], and some complexity results (e.g., the NP completeness of establishing the existence of a stable model for normal logic programs [4]) were sketched. This was also an occasion for discussing on *P vs NP* since some attendands were not aware of this open problem and of its relevance.

In the fourth lecture, programming in ASP was presended through simple examples. Moreover, installation of the required software in the laptops of the participants was carried on (see also Section 2.1). After examples of predicate

definitions on a family tree, the encoding of the classical problems of N-queens, magic square, wolf-goat-cabbage, and the three barrels problems were presented. A general introduction to planning problems, action description languages, and their ASP encodings has been presented [6].

Participants tested the executability of the proposed encodings with their laptops. As far as the magic square is concerned, a generate & test solution in C was also presented and tested. This is important since C is commonly perceived as "the fastest" language. Instead, in this way one can learn that if you don't have a good heuristics for the problem, ASP solution is not only nicer but also sensibly faster than a direct C encoding (just as example, for the 4×4 square, the C code finds the solution in 74 minutes, while the ASP solver clingo [8] takes only 1 second).

In the encodings we have decided of using just one built-in for defining a function (even if in particular cases, e.g. for Boolean functions, this can be done in a simpler way), namely:[4]

```
1{ cell(X,Y,1), cell(X,Y,2), cell(X,Y,3), cell(X,Y,4) }1 :-
        valuex(X), valuey(Y).
```

that forces the non-deterministic assignment of exactly one (the 1 on left states *at least*, the 1 on the right *at most*) value from 1 to 4 to each cell at coordinate (X,Y)—where the range of X and Y is stated by predicate `valuex` and `valuey`—and of its more compact but equivalent version:

```
value(1..4).
1{ cell(X,Y,V) : value(V)}1 :- valuex(X), valuey(Y).
```

The capability of ASP constraints of expressing universally quantified properties was also presented in detail with several examples. For instance:

```
:- valuex(X), valuey(Y), value(V), cell(X,Y,V), V < X + Y.
```

that can be read as "for all valuex X, forall valuey Y, and forall value V *it cannot be the case that* cell (X, Y) have a value $V < X + Y$."

A discussion on the syntax of aggregates was also made. Aggregates are rather easy to use and allow great programming expressiveness. However, since their semantics is based on "sets" rather than on "multisets" in GRINGO 4 (the last release of the clingo preprocessing/grounding tool), their use might lead to unexpected results. For instance, consider the following example:

```
dom(1..3).
p(1,1).  p(2,2).   p(3,3).
addall(S) :- S = #sum { Y : dom(X), p(X,Y) }.
```

The value of the argument of `addall` is in fact 6 as one might expect. In the following example, instead,

[4] This is also the idea used in [5] where constraint logic programming and ASP encodings of constraint satisfaction problems are compared.

```
dom(1..3).
p(1,1).  p(2,2).   p(3,2).
addall(S) :- S = #sum { Y : dom(X), p(X,Y) }.
```

one would expect `addall(5)`, while the result is `addall(3)`. Repetitions of 2 are removed. To fix the problem, one has to replace the last aggregate with

```
addall(S) :- S = #sum { Y,X : dom(X), p(X,Y) }.
```

The "`,X`" is not considered in the sum. But now the "`Y`" is taken either from the different pairs (2,2) and (2,3) and one gets the desired result. This has been experimentally verified to be counter-intutitive; therefore this preliminary example (or a similar one) is needed for explaining aggregate use.

The fifth and sixth lectures consisted in the explanation and verification of the encodings of other benchmarks, such as Hanoi Tower, Sam Lloyd's puzzle, Hammig code generation, Sudoku, and the encoding of a fussbal tournament with (a lot of) typical constraints. Finally, the encoding of two 3D puzzles, namely Braintwist and Rubik's cube were presented. For Rubik cube, the focus was on the simple $2 \times 2 \times 2$ version; together with the modeling, a visual rendering of the solver output, that animates the computed moves, was presented—see Figure 1. A propotype capable of providing a nice (animated) rendering of output of generic 2D games developed for this course is also presented and it will be discussed in Section 2.2.

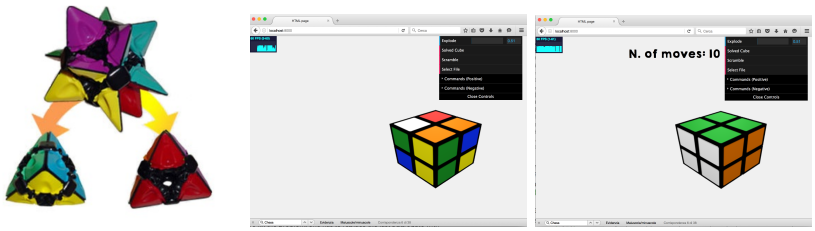

Fig. 1. Braintwist and the side-2 Rubik cube viewed with the animation tool.

The course was then suspended for two months to allow participants to work on some short lecture notes written by the course teacher, in order to extend them using a language tailored to high school students. New examples have been developed, new exercises and some more detailed explanations have been added. In future, the material will be organized in order to be split in lectures of two hours.

2.1 Software choice and installation

The choice of using the ASP solver clingo [8] is motivated by its well-known computational behaviour witnessed by the results of the various ASP competitions

and by its multi-platform (and free) distribution. As a matter of fact, teachers, students, school labs can use completely different (and, unfortunately, sometimes "old") operating systems (OS). However, while in a scientific context it is often sufficent to addredd th euser to a website and ask him/her to download the software (in this case `http://potassco.sourceforge.net/`), this is often not th ecase with high school students and, sometimes, high school lab technicians. Therefore, the installing instructions should be much more detailed. For instance, you might need to state: download the version for your operating system, once downloaded, unzip it and move it to the area where other programming languages are stored. Then you must be sure your operating system is aware of where clingo is located. Let us assume the path is `C:\ProgramFiles\clingo-v4.4\`: if you are using a windows machine, click the "start" button, go to control panel, system (or system and security, according to windows versions); click the "Advanced System Settings" link in the left column; in the "System Properties" window, click on the "Advanced tab", then click the "Environment Variables" button near the bottom of that tab. In the Environment Variables window highlight the "Path" variable in the "System variables" section and click the Edit button. Add or modify the path lines with the paths you want the computer to access. Each different directory is separated with a semicolon as shown below. `C:\ProgramFiles;...;\ProgramFiles\clingo-v4.4;` If you are using MACOS, you can add the alias (e.g. copy the line: `aliasclingo='$HOME/Tools/clingo-4.5.3-macos-10.9/clingo'`) in the file ".profile" that should be present in your home directory (the directory you can see when you open a terminal). If there is no such a file, please generate it. Linux users probably already know what to do (in case, just repeat the MACOS instruction in the .login file).

In all OS you should call clingo using a command-line instruction. Therefore you should run the "cmd" command in Windows and open a terminal in MACOS and Linux. We also experimented here that the command line use of PCs is no longer well-accepted outside the community of computer scientists.

Similarly we chose Geany [12] as editor (we found that some schools already used Geany for editing html pages and php scripts). Those already using other editors (win edt, emacs, etc) were of course allowed to use them. Instead participants willing to use notepad, wordpad, or even word/open office have been explicitly redirected to Geany.

We did not choose the nice ASPIDE [7] programming environment since the initial configuration can be justified by serious ASP programming but not by the simple examples of this course. Creating/Importing projects are too difficult notions to be explained at this stage, as well as warning messages such as "The external system is not specified or is missing".

2.2 A visual tool

We have already presented the Rubik cube visualization tool that has been prepared for the output of the ASP solver. We have also developed a Java tool, which is capable of visualizing in a pretty way the output returned by clingo for

problems concerning a chessboard or in general a 2×2 grid. For instance, this applies to chess-like problems, magic square, sokoban, Sam Lloyd's puzzle etc. The tool is able to deal either with static problems such as N-queens or dynamic problems (typically, planning problems) such as the Sam Lloyd's puzzle.

As far as the static version is concerned, in the current form we require that the programmer defines the following three predicates, that define the horizontal and vertical size of the board and the value to be put in each cell (X, Y).

```
xval(1..m).
yval(1..n).
cell(X,Y,VAL) :- xval(X), yval(Y), ... VAL ...
```

Then the output returned by the solver is processed and visualized as in Figure 2.

Fig. 2. Static view of the (first) ASP solution to the 10, 20, and 30 queens problems and to a 3×3 magic square problem.

For dynamic, planning problems a `time` predicate shoud be added and the time is the extra parameter for the `cell` predicate.

```
time(0..t).
cell(T,X,Y,VAL) :- xval(X), yval(Y), time(T), ... VAL ...
```

The solution is depicted for each time interval as shown in Figure 3.

As advanced feature the programmer can store some images and link them to the cell values: this way, generic pictures will be printed in cells instead of numbers and colors.

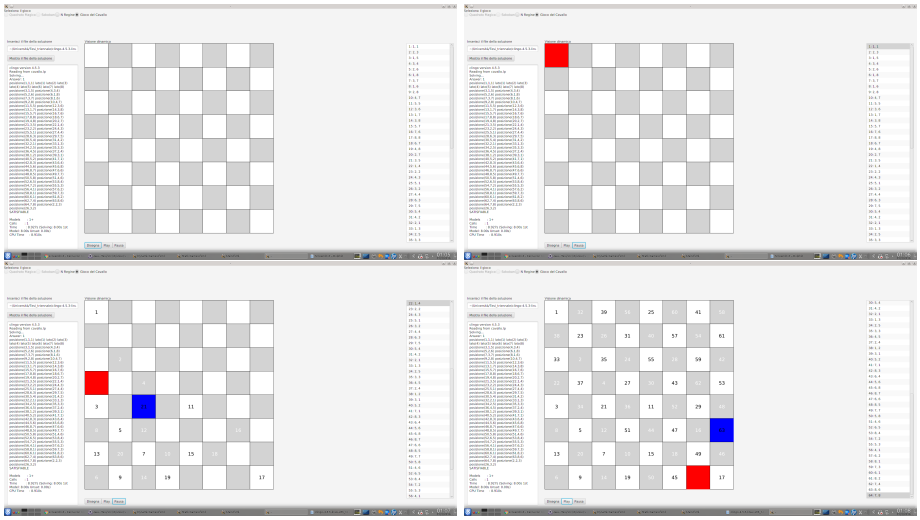

Fig. 3. Animated view of the ASP solution to the knight's tour problem.

2.3 Lecture Notes

Lecture notes have been organized following the sequence of ideas in slides presentation, save for the model theoretical results that are mostly omitted. The notion of (stable) model is presented at an intuitive level. A lot of simple examples have been added. The following one is interesting since its first encoding led us to a result that was considered counter intuitive. Imagine to define a program that models the throwing of two dices, in order to enumerate all combinations whose sum is 7. The initial program:

```
dice(1..6).
throw(X,Y):- dice(X), dice(Y).
```

obtains all results for `throw` (36 possibilities, from (1,1) to (6,6)). Then, using the experience of other encodings showed in classroom the following constraint is added (whose meaning is to remove all combinations that give a sum different from 7):

```
:- throw(X,Y), dice(X), dice(Y), X + Y != 7.
```

Instead of obtaining the desired result the solver reports: UNSATISFIABLE. This is correct of course, since all values for `throw` must be in any model due to the rule defining the predicate `throw`. This was judged counter intuitive by course participants, anyway. The correct program is the following.

```
dice(1..6).
throw(X,Y):- dice(X), dice(Y), X + Y = 7.
```

Although the issue emerges from a wrong understanding of the semantics of the definite part of the program that leads to a unique minimum model, some efforts should be spent in the lecture notes to clarify this point. The comparison of ASP ancoding and a C imperative encoding of the magic square problem have been also added in the lecture notes. Some simpler examples of declarative vs imperative code comparisons will be added as well. Lecture notes are available on-line from `www.dimi.uniud.it/dovier/DID/lpandgames.html` and will be updated in the future.

3 Conclusions and Future work

The three years experience presented has allowed us to prepare lecture notes (to be tested and refined next year with new students) and two visual tools, one instantiated on the 2D Rubik cube, andone more general for 2D grid problems. A visual tool for the Braintwist problem is also under development. We hope that, as a side effect, logical approaches to modeling will be spread in Italian high schools and that the next generation of students will be aware of logic-based modeling (and solving) techniques.

Acknowledgments. Alberto Policriti attended the whole course and provided a feedback after any lecture. His advices have been important for improving the successive lectures. Brain Twist and Rubik cube have been encoded by students Andrea Viel and Federico Igne, respectively. The pretty output interface was developed by student Gabriele Roncaglia. We thank the Univ. of Udine and of Trieste joint program Flash Forward 2, and the TID office of the University of Udine for their support to the various stages of the project described in the paper. We also thank Fabio Bove, Laura Candotti (ISIS Magrini-Marchetti, Gemona del Friuli) and Annalisa Nocino (Liceo Scientifico Statale G. Marinelli, Udine) for their active participation to the lectures.

References

1. Ancona, D., and Dovier, A. A theoretical perspective of coinductive logic programming. *Fundam. Inform. 140*, 3-4 (2015), 221–246.
2. Apt, K. R. *From logic programming to Prolog.* Prentice Hall International series in computer science. Prentice Hall, 1997.
3. Cooper, B., and van Leeuwen, J. *Alan Turing: His work and Impact.* Elsevier, 2013.
4. Dantsin, E., Eiter, T., Gottlob, G., and Voronkov, A. Complexity and expressive power of logic programming. In *Proceedings of the Twelfth Annual IEEE Conference on Computational Complexity, Ulm, Germany, June 24-27, 1997* (1997), IEEE Computer Society, pp. 82–101.
5. Dovier, A., Formisano, A., and Pontelli, E. An empirical study of constraint logic programming and answer set programming solutions of combinatorial problems. *J. Exp. Theor. Artif. Intell. 21*, 2 (2009), 79–121.

6. Dovier, A., Formisano, A., and Pontelli, E. Perspectives on logic-based approaches for reasoning about actions and change. In *Logic Programming, Knowledge Representation, and Nonmonotonic Reasoning - Essays Dedicated to Michael Gelfond on the Occasion of His 65th Birthday* (2011), M. Balduccini and T. C. Son, Eds., vol. 6565 of *Lecture Notes in Computer Science*, Springer, pp. 259–279.
7. Febbraro, O., Reale, K., and Ricca, F. ASPIDE: integrated development environment for answer set programming. In *Logic Programming and Nonmonotonic Reasoning - 11th International Conference, LPNMR 2011, Vancouver, Canada, May 16-19, 2011. Proceedings* (2011), J. P. Delgrande and W. Faber, Eds., vol. 6645 of *Lecture Notes in Computer Science*, Springer, pp. 317–330.
8. Gebser, M., Kaminski, R., Kaufmann, B., and Schaub, T. Clingo = ASP + control: Preliminary report. *CoRR abs/1405.3694* (2014).
9. Gelfond, M., and Lifschitz, V. The stable model semantics for logic programming. In *Logic Programming, Proceedings of the Fifth International Conference and Symposium, Seattle, Washington, August 15-19, 1988 (2 Volumes)* (1988), R. A. Kowalski and K. A. Bowen, Eds., MIT Press, pp. 1070–1080.
10. Newell, A., Shaw, C., and Simon, H. Chess playing programs and the problem of complexity. *IBM Journal of Research and Development 4*, 2 (1958), 320–335.
11. Niemelä, I. Logic programs with stable model semantics as a constraint programming paradigm. *Ann. Math. Artif. Intell. 25*, 3-4 (1999), 241–273.
12. Tröger, E., Treleaven, N., Lanitz, F., Wendling, C., and Brush, M. Geany. A fast, light, GTK+ IDE, 2016.