Answer Set Programming and Declarative Problem Solving in Game AIs

Davide Fuscà, Stefano Germano, Jessica Zangari,
Francesco Calimeri, and Simona Perri

Dipartimento di Matematica e Informatica, Università della Calabria, Italy
{ddfusca,stefanogermano0,jessica.zangari.90}@gmail.com,
{calimeri,perri}@mat.unical.it

Abstract. Designing and implementing AI in games is an interesting, yet complex task. This paper briefly presents some applications that make use of Answer Set Programming for such a task, and show some advantages of declarative programming frameworks against imperative (algorithmic) approaches while dealing with knowledge representation and reasoning: solid theoretical bases, no need for algorithm design or coding, explicit (and thus easily modifiable/upgradeable) knowledge representation, declarative specifications which are already executable, very fast prototyping, quick error detection, modularity.

Keywords: Declarative Programming, Answer Set Programming, Artificial Intelligence, Computational Logic, Knowledge Representation and Reasoning

1 Introduction

This work presents some Artificial Intelligence applications designed and implemented during the Course of Artificial Intelligence in the context of the Computer Science Bachelor Degree at University of Calabria, Italy[1]. The aim of each project was to study and reproduce the behavior of a skilled player of some "classic" games. In particular, the explicit knowledge and the reasoning modules have been implemented by means of Answer Set Programming (ASP) techniques, and the projects can be seen as a nice showcase of features, power and advantages coming from the use of ASP itself. In the following, after a brief introduction to ASP, we will illustrate the above mentioned projects, especially focusing on the approaches adopted for the implementation of the AIs.

2 ASP and Declarative Problem Solving

Answer Set Programming (ASP) [3,11] became widely used in AI and is recognized as a powerful tool for knowledge representation and reasoning (KRR),

[1] http://www.mat.unical.it/ComputerScience

especially for its high expressiveness and the ability to deal also with incomplete knowledge [3]. The fully declarative nature of ASP allows one to encode a large variety of problems by means of simple and elegant logic programs. The semantics of ASP associates a program with none, one, or many answer sets, each one corresponding one-to-one to the solutions of the problem at hand.

For instance, let us consider the well-known NP-complete *3-Colorability* problem: given a graph, decide whether there exists an assignment of one out of three colors to each node, such that adjacent nodes never have have the same color. Each instance can be represented by a set of facts F over predicates $node(X)$ and $arc(X, Y)$. The following program, in combination with F, computes all 3-Colorings (as *answer sets*) of the graph represented by F.

$$\begin{array}{ll} r_1: & color(X, red) \mid color(X, green) \mid color(X, blue) \leftarrow node(X). \\ r_2: & \leftarrow \ color(X_1, C), color(X_2, C), arc(X_1, X_2). \end{array}$$

Rule r_1 expresses that each node must be colored either red, green, or blue; due to the minimality of the answer sets semantics, a node cannot be assigned more than one color. The integrity constraint r_2 enforces that no pair of adjacent nodes (connected by an arc) is assigned the same color.

The paradigm adopted above is one of the most commonly used among ASP programmers, and is referred to as the "Guess&Check" methodology [8]. An extension to this methodology is the so called "Guess/Check/Optimize" [4].

In summary, an ASP program that matches GCO features 3 modules:

- **Guessing Part** defines the search space (by means of Disjunctive Rules)
- **Checking Part** (optional) checks solution admissibility (by means of Integrity Constraints)
- **Optimizing Part** (optional) specifies a preference criterion (by means of Weak Constraints)

In the latest years many efforts have been spent in order to obtain solid and efficient systems supporting ASP, and a number of modern systems are now available (see [5] for a pretty comprehensive list and a more detailed bibliography about ASP). In the present works we used DLV [12], a deductive database system supporting Disjunctive Datalog properly enriched with weak constraints (to express optimization problems), aggregates (to better deal with real data and applications), queries and other language extensions.

It is worth noting that many other logic formalisms out there can explicitly represent an agent's knowledge, and allow one to accomplish the AI jobs herein discussed; one of the most widely known is Prolog, that has already been employed before in AI games [1,6,7,10]. However, we wanted to specifically follow a fully declarative approach. Prolog, for instance, requires to know the resolution algorithm while writing a program, while in ASP the order of rules within a program, as well as the order of subgoals in a rule, is irrelevant. This paper does not aim at specifically discussing differences between ASP and other formalisms; for further details about such topics we refer the reader to the extensive existing literature.

3 Applications

3.1 General Architecture

The applications herein presented share the same basic architecture (see Figure 1), which can be seen as consisting of three layers: the *core*, the *ai* and the *gui*. The *core* layers connects the *ai* layer with the *gui* layer: it manages the game via the *gui* while providing the *ai* with proper information and getting in turn the (hopefully "right") actions to be performed. The ad-hoc *gui* layer allows a user to play the game against the machine in an intuitive way.

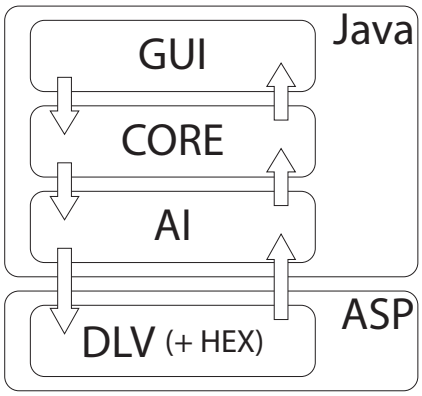


Fig. 1: General Architecture

Indeed, the architecture hides a more general framework for ease the development of applications in which AI is crucial. This is not only the case of games, but also the scenario of relevant practical problems in presence of incomplete or contradictory knowledge. The *ai* module is uncoupled from the *core* and the *gui*, that can be designed independently from the AI; in addition, the latter can be gradually improved (or easily interchangeable).

In the present setting, the *ai* module is based on ASP and uses the DLV system as the actual ASP solver.

The applications are implemented in Java, and in particular for the 2D visualization of the game we used the Java GUI widget toolkit Swing.

3.2 Connect Four

Game Description Connect Four is played by two opponents on a vertical 7×6 rectangular board. The players fill the board by dropping 1 disk in turn from the top to the bottom of the board: if a disk is dropped in a column, it falls down onto the lowest unoccupied position within it. The winner is the first player who gets four of her disks in a line, connected either horizontally, vertically, or diagonally. As common in board games, we will refer to the "White" player as the one who begins the game, and to the "Black" player as the other one.

Implementing the Artificial Intelligence Connect four was mathematically solved in 1988 by Victor Allis[2], showing if both players play perfectly, the White one is always able to win if she begins from the middle column, while starting from another column enables the second player to always force a drawn.

Three levels of AI have been implemented: the hardest one implements the rules of perfect play emerged from Allis' work, while the easiest relies on some classical heuristic strategies, and the intermediate mixes some basic Allis' rules with some common strategies. Furthermore, the White and Black strategies sensibly differ. Intuitively, the White player needs to *keep* the advantage deriving from the simple fact that she starts the game, while the other has heavier burden: she needs to *turn* the game on his advantage defending from the natural ongoing of the game in favour of the opponent and trying to exploit the possible "errors" made by the adversary. This has been straightforwardly accomplished by developing different ASP programs, each one based on the "Guess/Check/Optimize" technique:

- **WhiteAdvanced**, **BlackAdvanced**: highest level for both players;
- **WhiteIntermediate**, **BlackIntermediate**: intermediate level;
- **Easy**: easiest level; in this case we can observe the consequences of undifferentiated strategies.

A further ASP module, called **CheckVictory**, is used to check if one of the player is the winner each time that player makes a move. In order to provide the reader with an idea about the way ASP is employed here, we show next some ASP rules shared by all the modules; full ASP encodings are available online (see Section 4).

Guess : $selectedCell(R,C) \mid notSelectedCell(R,C) \leftarrow playableCell(R,C).$
Check 1 : $\leftarrow \neg\#count\{R,C : selectedCell(R,C)\} = 1.$
Check 2 : $\leftarrow \neg selectedCell(5,3), playableCell(5,3).$
Optimize 1 : $:\sim threat(A,R,C), \neg selectedCell(R,C), playableCell(R,C), me(A).[1:5]$
Optimize 2 : $:\sim threat(A,R,C), \neg selectedCell(R,C), playableCell(R,C), opponent(A).[1:4]$

The **Guess** rule generates all possible split of the "playable" cells (R,C standing for row and column, respectively) into two sets, the selected for the next move, and the rest. But the player can occupy only one cell at each turn: the **Check 1** rule enforces this. This is a basic player, with no particular "intelligent" behaviour.

The **Check 2** provides us with a first strategic glimpse: if cell (5,3) is still "available"[2], the player has to occupy it: this is known to give the player an advantage. The last two rules are part of the "optimize" task: the first is an

[2] A cell is "playable" if it is not occupied yet and stands in the lowest row, or if the cell below it in the same column is already occupied.

attack rule, while the second is a defence one. A "threat" for a player A is a cell which, if taken by player A, connects four of her disks: *threat*(A,R,C) means that the cell (R,C) is a threat for player A. If the player has a threat, she should occupy the threat cell as soon as possible: this is "pushed" by the **Optimize 1** rule. Similarly, rule **Optimize 2** "pushes" the player to occupy a cell if it is a threat for the opponent. The two optimization statements have different weights: in case the player can choose among an attack or a defence move, the rational behaviour is to perform an attack. Actually, rule **Optimize 1** has the greatest value among all the other "optimize" rules describing different strategies: if victory is just a move away, just make that move!

Intuitively, the various AI program differs especially in their optimization. The aim is to depict different scenarios by means of different strategies: this made the game very well suited to be analysed from an AI perspective. It is of clear interest to compare the different AI levels in a game between two artificial players, assessing the "perfect" against the heuristic-based strategies; in addition, we can assess the AIs against human players, who can provide a wide range of different strategies. A further positive aspect, is that by avoiding a brute-force search approach in favour of a *knowledge-based approach*, not only changes are easy to implement and assess, but different *styles* can also be easily described and actually implemented.

3.3 Reversi

Game Description Reversi is a strategy board game with 64 game pawns, each one featuring both a black and a white side. Differently from the typical assumption, Black plays first and places a disc with the black side up. At each move, there must exist at least one straight (horizontal, vertical, or diagonal) occupied line between the new piece and another black piece, with one or more contiguous white pawns between them. Once the piece is placed, black turns over all white discs lying on such straight line between the new piece and any anchoring black pawns. Similarly does the White. If a player does not have a legal move available at any time, she must pass, and her opponent plays again; if neither player has a legal move, the game ends. The winner is the player with the higher number of pawns of his own color on the board.

Implementing the Artificial Intelligence The intelligence is described by means of logic rules according to the GCO (Guess/Check/Optimize) technique. At each turn, some facts representing the board state are coupled with the logic program, which has been conceived so that answer sets represent the moves: strong constraints "drop" any answer set representing non-valid moves, while weak constraints "select" the best one.

We started from a basic game manual[3]. In order to create different levels of AIs we selected some strategies with different features, each one represented by

[3] available at `http://www.fngo.it/corsobase.asp` on the FNGO (Italian Federation of the Othello Game) website

a set of rules that can be added incrementally: each level uses all the strategies from of the previous, plus some more sophisticated. The different AIs can be summarized as follows:

- **None**: the simplest one; trivially chooses a valid move;
- **Basic**: tries to maximize the number of pawns eaten with each move;
- **Medium**: adds the strategy of "corners and stable pawns";
- **Hard**: adds the strategy of "border pawns and walls".

Five different ASP modules helps at representing such AIs. One module models the Guess/Check, one models the current state of the board (plus some other useful pieces of information), and we have one additional module for each strategy. The artificial player can change its behaviour by simply running the first two modules along with the one related to the desired strategy. We present next some ASP rules featured by the mentioned modules; full ASP encodings are available online (see Section 4).

Guess	$selectedCell(R, C) \mid notSelectedCell(R, C) \leftarrow validCell(R, C).$
Check	$\leftarrow \neg \#count(R, C : selectedCell(R, C)) = 1.$
Optimize	$:\sim notSelectedCell(R, C), cornerPawn(R, C).\ [1:15]$

Similarly to the previous case, the two rules (**Guess**) and (**Check**) select exactly one cell, among all legal moves. The rule (**Optimize**) is a Weak Constraint that expresses the fact that, if possible, the player should choose the corner cells at the corners (these are known to be "strategic positions" in the game); this is a nice example of how easy is to incorporate explicit knowledge of a domain, as it might be provided by an expert.

The Reversi Action Addon We have designed and implemented also an add-on (the *Reversi Action Addon*) for the *Action Plugin* of the DLVHEX solver [9], thus making the system able to play an online version of Reversi[4]. It employs the same logic rules herein described, with some adaptations for the sake of compatibility with the DLVHEX solver and the online game. The Add-on is completely autonomous, with no need of human intervention (apart from the boot). By means of Javascript and Perl scripts it logins into the website, recognizes the game status and makes its moves. It's also able to wait the opponent's move and to understand when the game is over.

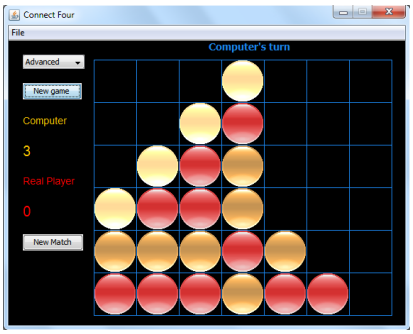
4 Conclusion

In this paper we presented some applications that make use of the capabilities of ASP in the design and implementation of the AI, which is, in these cases, the most complex (and interesting) part. We implemented some "classic" strategies,

[4] Online game available at `http://www.yourturnmyturn.com`

typically difficult to implement when dealing with the imperative programming, in a rather simple and intuitive way. Moreover, we had the chance to test the AI without the need for rebuilding the application each time we made an update, thus observing "on the fly" the impact of changes: this constitutes one of the most interesting features granted by the *explicit* knowledge representation. In addition, we developed different versions of the AIs, in order to show how easy is to refine the quality or to generate different strategies or "styles"; and these include also non-winning, *human-like* behaviors.

The games herein presented can be downloaded at `https://www.mat.unical.it/calimeri/files/AIgames/PAI2013/games.zip`; the packages contain the full ASP programs herein sketched.



(a) Connect 4 Screenshot

(b) Reversi Screenshot

References

1. CGLIB- A Constraint-based Graphics Library for B-Prolog. `http://probp.com/cg_examples.htm`
2. Allis, L.V.: A knowledge-based approach of connect-four. Vrije Universiteit, Subfaculteit Wiskunde en Informatica (1988)
3. Baral, C.: Knowledge Representation, Reasoning and Declarative Problem Solving. Cambridge University Press (2003)
4. Buccafurri, F., Leone, N., Rullo, P.: Strong and Weak Constraints in Disjunctive Datalog. In: Dix, J., Furbach, U., Nerode, A. (eds.) Proceedings of the 4th International Conference on Logic Programming and Non-Monotonic Reasoning (LPNMR'97). Lecture Notes in AI (LNAI), vol. 1265, pp. 2–17. Springer Verlag, Dagstuhl, Germany (Jul 1997)
5. Calimeri, F., Ianni, G., Ricca, F.: The third open answer set programming competition. Theory and Practice of Logic Programming 1(1), 1–19 (2012)
6. Clark, K.L.: Negation as failure. In: Logic and data bases, pp. 293–322. Springer (1978)
7. Colmeraner, A., Kanoui, H., Pasero, R., Roussel, P.: Un systeme de communication homme-machine en francais. Luminy (1973)

8. Eiter, T., Faber, W., Leone, N., Pfeifer, G.: Declarative problem-solving using the dlv system. In: Logic-based artificial intelligence, pp. 79–103. Springer (2000)
9. Fink, M., Germano, S., Ianni, G., Redl, C., Schüller, P.: Acthex: Implementing hex programs with action atoms. In: 12th International Conference on Logic Programming and Nonmonotonic Reasoning (LPNMR), LNAI 8148. pp. 317–322 (2013)
10. Finnsson, H., Björnsson, Y.: Simulation-based approach to general game playing. In: AAAI. vol. 8, pp. 259–264 (2008)
11. Gelfond, M., Lifschitz, V.: Classical Negation in Logic Programs and Disjunctive Databases. New Generation Computing 9, 365–385 (1991)
12. Leone, N., Pfeifer, G., Faber, W., Eiter, T., Gottlob, G., Perri, S., Scarcello, F.: The DLV System for Knowledge Representation and Reasoning. ACM Transactions on Computational Logic 7(3), 499–562 (Jul 2006)