Informatics in Education, 2009, Vol. 8, No. 1, 85–100

© 2009 *Institute of Mathematics and Informatics, Vilnius*

Teaching Artificial Intelligence and Logic Programming in a Competitive Environment

Pedro RIBEIRO
CRACS & INESC-Porto LA
DCC – Faculdade de Ciências, Universidade do Porto
R. Campo Alegre, 1021/1055, 4169-007 Porto, Portugal
e-mail: pribeiro@dcc.fc.up.pt

Hugo SIMÕES
LIACC – Artificial Intelligence and Computer Science Laboratory
DCC - Faculdade de Ciências, Universidade do Porto
R. Campo Alegre, 1021/1055, 4169-007 Porto, Portugal
e-mail: hrsimoes@dcc.fc.up.pt

Michel FERREIRA
Instituto de Telecomunicações
DCC – Faculdade de Ciências, Universidade do Porto
R. Campo Alegre, 1021/1055, 4169-007 Porto, Portugal
e-mail: michel@dcc.fc.up.pt

Received: October 2008

Abstract. Motivation plays a key role in the learning process. This paper describes an experience in the context of undergraduate teaching of Artificial Intelligence at the Computer Science Department of the Faculty of Sciences in the University of Porto. A sophisticated competition framework, which involved Prolog programmed contenders and game servers, including an appealing GUI, was developed to motivate students on the deepening of the topics covered in class. We report on the impact that such a competitive setup caused on students' commitment, which surpassed our most optimistic expectations.

Keywords: education, artificial intelligence, logic programming, teaching frameworks, competitive learning.

1. Introduction

Teaching Logic Programming (LP), and its favourite application subject of Artificial Intelligence (AI), has a strong tradition at the Computer Science Department of the Faculty of Sciences in the University of Porto (DCC-FCUP). The department was created by early users of the LP paradigm, in areas such as natural language processing (Porto and Filgueiras, 1984; Filgueiras, 1987), and pioneer implementors of LP systems, such as EMAS Prolog and C-Prolog (Roy, 1994). This tradition lead to the development of one

of the most efficient and widely used Prolog systems, Yap Prolog (Costa, 2008), and established at University of Porto a very strong research group on LP.

Motivation plays a very important role in the process of learning (Bergin and Reilly, 2005). Teaching LP is no exception and one needs precisely to motivate the students. One way of doing that is to use competition to promote the desire of being better (Wallace and Margolis, 2007). There is an enduring relationship between AI and games, which make these a very good vehicle to teach the subject. That has been done for example in (Hingston *et al.*, 2006), but also in the context of another computer science discipline (Lawrence, 2004).

Using a similar line of thought, this paper describes an experience in the context of the undergraduate teaching of AI, when the three co-authors of this paper were jointly involved in the lecturing of the subject at DCC-FCUP. We developed a sophisticated competition framework, which involved Prolog programmed contenders, game servers and code obfuscators, and an appealing Java graphical user interface (GUI), with the goal of improving students motivation on the deepening of the topics covered in class. At the beginning of the course we presented students a practical assignment which consisted in programming an intelligent player of a particular board game. The assignment grade was a substantial component of the overall grade for the AI course. More importantly, the assignment grade was to be determined by the ranking obtained in an all-against-all competition among student programs, where we would also include our own programs, some working as level definers, and one representing our best effort in trying to program an unbeatable player.

The response obtained from students was impressive, surpassing our most optimistic expectations. Suddenly, we had students searching and reading literature covering much more advanced topics than what was covered in class. We had students worried in being able to test their programs against its competitors, while protecting their code *"copyright"*, a concern usually associated with teachers rather than with students. The level of enthusiasm was such that we were asked to improve the GUI, which animated the competition, to include the customization of the pieces *"skins"*, which would be supplied by the students, as the simple red and blue pieces were considered inappropriate for such highly competitive programs, proudly named as *Black Knight*, *Khan* or *Owl*!

The remainder of this paper is organized as follows. Section 2 presents the game that inspired the competition. Section 3 describes the competition framework, presenting the game server, the graphical user interface and the Prolog code encryptor. Section 4 presents the actual competition between the participating programs, its qualifying premises and competitive evaluation criteria, and summarizes the results of the competition. Section 5 describes the winning programs, focusing on the overall winner, while also describing the interesting strategies implemented by notable programs. Section 6 reports on the feedback we received from the students. Finally, Section 7 concludes.

2. The Ataxx Game

For our educational experience, we wanted to use a game that would provide simplicity of rules, but at the same time give plenty of opportunities to solve the problem in interesting

ways. It was also our goal to choose a game that would not be one of the "classical" ones (like chess), in order to provide a fresh start to everyone and equal chances of thinking the strategies to play it. We decided to use Ataxx, a board game, and we will now explain its origins and rules.

The Ataxx game was invented in 1988, in England, by David Crummack and Craig Galley, with the original name of *Infection* (Beyrand, 2005). They wanted to make a game that would work better on a computer than on a traditional board and they got their inspiration from other known games such as Reversi. It was initially programmed for Amiga, Commodore 64 and Atari ST platforms. Although this particular implementation was never commercially released, its concept was so interesting that its company quickly decided to use it on PC (with the name *Spot*), and on arcade games (with the name *Ataxx*). The game was popularized in 1992 when it was featured as one of the most difficult puzzles in the 7th Guest game, with the name *Microscope* or *Show of Infection* (7th Guest was a huge success on its time). After that, the game had many versions, with many names (such as *SlimeWar* or *Frog Cloning*) and variations, but its most common name became Ataxx, which was derived from the word *attack*.

Ataxx is played in a 7×7 grid board by two opponents that take turns to play, each one playing with pieces of a different color. At the start of the game, there are two pieces of each player, situated on opposite corners of the board.

In its turn, a player must move one of its pieces to an empty square. He can move the piece to one of its adjacent squares, and the piece is replicated, in the sense that a new one appears in the new position. In alternative, he can make a piece jump to a free position that is at a distance of two squares from the original one (with no replication). Fig. 1 shows the possibilities for a piece move.

After a move, all the opponent pieces that are neighbors of the destination square swap their color to match the piece that moved. See Fig. 2 for an example of two different moves and their respective consequences.

If a player does not have any possibility for moving, it has to pass its turn to the other player. A game ends when the board is full, when a player has no pieces or when a board configuration is repeated three times by the same player. The player with the higher number of pieces at the end wins the game.

There exist several variations on these rules (boards with different sizes or shapes, blocked squares, etc), but these are the original and basic ones, which we decided to use on our experiment.

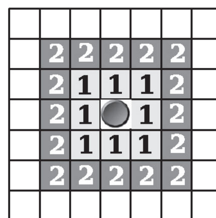

Fig. 1. Ataxx possibilities for a move. The piece in the centre can replicate itself to the 1-squares, and it can jump to the 2-squares.

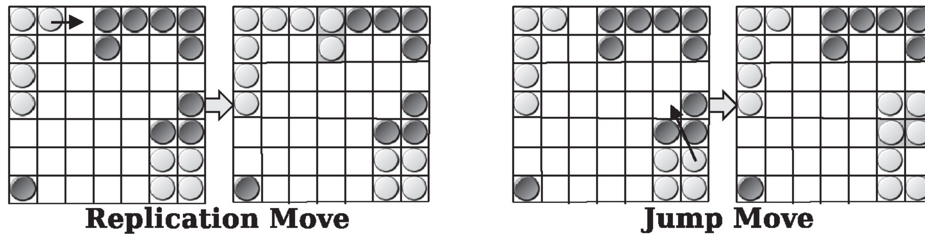

Fig. 2. Two examples of Ataxx possible moves.

Although with very simple rules, Ataxx provides a rich and interesting environment for gameplay. Note that on a single move, the score can change in a very large way, because eight new pieces can be added to the player that moved, and these same eight new pieces also "disappear" from the other player's score. This makes the game very unpredictable and completely dynamic, with lots of changes. Typically, the last few moves are really important and decisive.

3. Competition Framework

Our main goal was of course to teach AI and LP in Prolog but we wanted to do it in a way that would really motivate the students. Having chosen a specific game, we needed to present it an attractive and usable way. We wanted to create a completely functional framework for playing the game, both as a human and as an artificial agent. However, we could not forget to do that in an efficient way, since as you will see on Section 4 we will have to make a large number of games.

We decided to use a modular approach, that also permitted us to divide the programming effort among us, to present the students with the fully functioning framework. We have an Ataxx server component, that is responsible for implementing the rules and interacting with the programs, and we also have a graphical user interface that the students can use to visualize and play the games, although they can choose to directly use the server in a textual way. Fig. 3 gives an overview of the architecture we used, and we proceed describing each of the components of the architecture in more detail. All of the tools can be downloaded and experimented in (Ribeiro *et al.*, 2008).

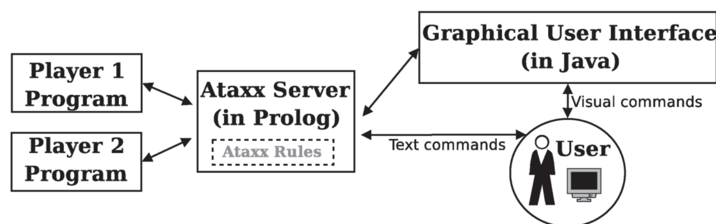

Fig. 3. Our Ataxx framework general architecture.

3.1. *The Ataxx Server*

The Ataxx server is the main component of all the framework. It is responsible for implementing the rules and for providing an interface to interact with the students' programs. Since the whole idea was to make the agents in Prolog, we also opted to write this server in the same language.

Basically, the server functions in a textual way, interacting with the user using textual input and output. The user can start, run and stop a game. The server offers a limited text depiction of the board and more importantly it provides a way to run the game in batch (unattended) mode, giving an automatic and fast way to play a game. The server is also responsible for implementing the Ataxx rules, making sure that all moves are valid and determining when the game is finished. Besides all this, the server also has embedded a random agent, that we can use to have a very basic opponent to test our programs.

To run a game, the server needs to have access to the two agents' Prolog files. Is was established that those agents should always have a top level predicate with the format `main(+Board,+Timeleft,-Move)`, which will be called by the server. The board is given as a list of lists representing a matrix, with the characters `'x'` for the current player pieces, `'o'` for the opponent pieces and `'-'` for empty squares. The server is responsible for changing the pieces in order for the current player to always have `'x'` as his pieces. This was made to facilitate the students' task. `Timeleft` is an integer representing the total milliseconds left for all the agent's remaining moves. `Move` is the output of the program representing a move, and it should be an atom of the form `'ABCD'`, where `(A,B)` are the coordinates of the piece's original position and `(C,D)` is the destination square.

Note that the agents do not have access to any kind of memory between moves. Each time they are called again as if they were in the start of the game, but with the new modified board. The server guarantees that the programs only use at maximum the remaining available time.

3.2. *Graphical User Interface*

Since we wanted an attractive way to present the game, we decided to create a complete GUI for our Ataxx experience. This interface is modular, and can be optionally attached to the server, since it is independent from the game itself. We used Java Swing (Swing) for the widgets and we included a clickable interface to be used by humans playing the game. In this way, it is possible to have computer vs. computer, human vs. computer and human vs. human matches, which can be used by the students to become more familiar with the game and devise the strategies that they will implement. Fig. 4 shows a screenshot of the GUI.

The GUI includes animations for the piece captures and provides ways to automatically run a full game, advance turn by turn and undo or redo moves. It also implements another possibility that was deemed very useful for the students: the use of game logs. Students can save and load entire or partial games. For example, they can run games

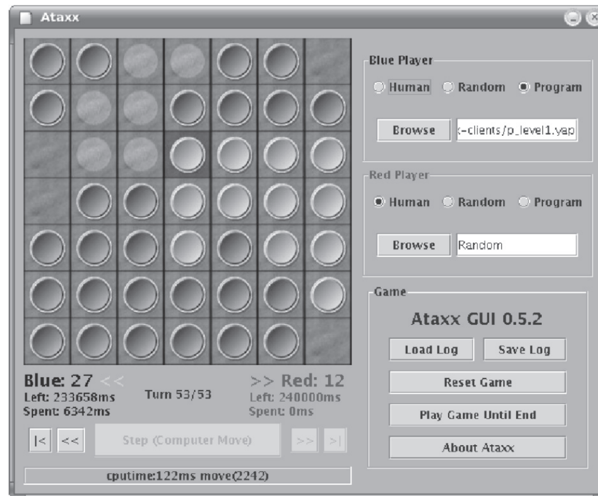

Fig. 4. A screenshot of our Ataxx graphical user interface.

in batch mode and then check how the games were by loading the logs. And since the log format we choose was a readable textual one, they could even prepare special board positions to evaluate their programs.

3.3. *The Prolog Code Obfuscator*

It soon became clear for all teams that testing agents against the embedded random agent or against their own previous agents was not enough to have an estimate on how an agent would perform against other agents in the official competition. Furthermore, students wanted to protect their code from being copied or readily understood by some other team (including teachers). In order to solve this problem, we decided to develop another tool: a Prolog code obfuscator.

Fig. 5b shows the result of obfuscating the QuickSort program of Fig. 5a using our ProTuP (**Pro**log **T**o **u**nreadable **P**rolog) obfuscator.

Regardless of whether it is possible to protect an obfuscated code from being understood, we just needed to make such attempt difficult enough to be discouraging, in addition to preventing the original source from being retrieved through the obfuscated code.

ProTuP changes a given Prolog source by stripping comments, renaming variables, removing unnecessary syntax, renaming user-defined predicates (except 'main') and randomly sorting the position of these predicates.

The first three types of changes can be done automatically using Prolog ISO predicate *read/1*[1], resulting in variables getting renamed to tokens of the form `_n`, where *n* is a natural number. To further reduce readability, the user-defined predicates are renamed to tokens of the form `'''_n'''`.

[1] For extra control we have actually used Prolog ISO predicate *read_term/2*.

```
partition([], _, [], []).
partition([X|Xs], Pivot, Smalls, Bigs) :-
    (   X @< Pivot ->
        Smalls = [X|Rest],
        partition(Xs, Pivot, Rest, Bigs)
    ;   Bigs = [X|Rest],
        partition(Xs, Pivot, Smalls, Rest)
    ).

quicksort([], []).
quicksort([X|Xs], Ascending) :-
    partition(Xs, X, Smaller0, Bigger0),
    quicksort(Smaller0, Smaller),
    quicksort(Bigger0, Bigger),
    append(Smaller, [X|Bigger], Ascending).
```

a) Non-obfuscated

```
'''_0'''([],[]).
'''_0'''([_3505|_3506],_3507):-
    '''_1'''(_3506,_3505,_3511,_3512),
    '''_0'''(_3511,_3518),
    '''_0'''(_3512,_3522),
    append(_3518,[_3505|_3522],_3507).

'''_1'''([],_1359,[],[]).
'''_1'''([_1727|_1728],_1729,_1730,_1731):-
    _1727@<_1729->
    _1730=[_1727|_1742],
    '''_1'''(_1728,_1729,_1742,_1731);
    _1731=[_1727|_1742],
    '''_1'''(_1728,_1729,_1730,_1742).
```

b) Obfuscated

Fig. 5. QuickSort in Prolog.

When comparing Fig. 5a and Fig. 5b notice that the order in which the predicates are defined is inverted and also that the parentheses surrounding the body of the second *partition* clause have been removed.

Imagining the result of obfuscating the code of Fig. 5a together with a few dozens of other user-defined predicates, we believe ProTup met its goals.

4. Course Assignment and Competition

The Ataxx assignment was integrated in an one semester AI course, taught in the third year of a Computer Science undergraduate degree. Before entering it, students already have solid groundwork on Computer Science foundations as a whole. They are already exposed to imperative and functional programming paradigms, covering all steps in developing a program, from data abstraction to algorithm design. The course is however their first introduction to LP. Although some AI related techniques may have already been approached, this course is also their first real organized approach to the AI field, surveying the main concepts and methods used.

Each week of the AI course was composed of 3.5h of theoretical classes and an 1h practical class on a computer lab. A total of 135 students frequented the course and each practical lab had an average of 17 students. The Ataxx assignment in itself accounted for 20% of the final grade in the course. In the previous years this AI course had no extensive practical assignment (only 1h in class Prolog programing evaluations). The amount of students that effectively succeeded in the course was reduced and one of the main complaints present on the surveys made was that no real practical experience on AI was obtained. Our main goal was therefore to motivate the students, both in LP and AI, giving them an hands on competitive approach on a practical AI task. We felt that this could give a decisive positive contribution on the learning experience, improving both the results obtained and the effective material learned.

Several theoretic and practical classes were dedicated to the the assignment and the students had a time span of about two months to build their agents. Some Ataxx related predicates that could potentially be used on the agent were made available for automatic evaluation using Mooshak system (Leal and Silva, 2003). Basically a set of inputs was given to the students code and the system would automatically tell them if the output produced was correct. Students could submit their code as many times as they wish. An example predicate available on Mooshak was to process an Ataxx move, effectively producing the obtained game board after it.

The Ataxx agent was to be developed by groups of two or three students. The requirements were very simple: the programs should be implemented in Prolog, having a top-level predicate `main/3` which would be invoked by the Ataxx server, and should complete every game using 4 minutes of CPU time. Taking longer than this would immediately result in the loss of the game, by the maximal margin, 0–49. Invalid moves passed to the server also result in losses by the same maximal margin. There was no restriction on the size of the programs.

The competition would be run on AMD Sempron(tm) 2800+ machines, with 1Gb of RAM, featuring 3964.92 Bogomips, present in the Department labs.

We had a total of 46 programs submitted to evaluation, amounting to a total of 104 students. This means that a very significant amount of 77% of the total number of students enrolled in the course responded to the assignment. All these 46 programs participated in a preliminary qualification round where they would play a total of 18 games against three level defining programs, named *Random*, *Greedy* and *Level1*. The *Random* player implemented what the name suggests, randomly selecting a valid move, without any strategy. Interestingly, one group of students, *RPR*, succeeded in developing a validly playing program that was able to lose 5 of the 6 games played against *Random*, an effort requiring some engineering, but unfortunately in the wrong direction. *Greedy* implemented a greedy strategy, choosing a move that maximized the instant difference between its pieces and the opponent pieces. This program was non-deterministic, in the sense that equally maximizing moves were arbitrarily chosen, with the single condition that non-jumping moves were always preferred. *Level1* implemented a minimax algorithm of depth 1, i.e. one level more than the *Greedy* program, choosing the move that minimized the opponent maximization. In this qualification round each students' program played 6 games against each of the *Random*, *Greedy* and *Level1*, 3 as first player and 3 as second player. This qualification round determines the distribution of the students' programs in three groups, as follows:

- *Premier League*, consisting of the programs that do not lose in the overall confront against each of the 3 level defining programs.
- *League of Honor*, consisting of the programs that did not qualify for the *Premier League* but win more games than lose in the qualification round.
- *Unqualified*, consisting of all the other programs.

Programs in the Premier and Honor leagues play against all the programs in their leagues, a total of four times with each opponent, two as first player and two as second player. The grades are determined by the final ranking after the competition, and are

Table 1
Qualification round results

Program	**Total**		**Random (W – L)**	**Greedy (W – L)**	**Level1 (W – L)**	**Qualification**
	W – L	**Goal Avg.**				
genocide	18 – 00	+594	6 – 0	6 – 0	6 – 0	Premier League
Toninja	18 – 00	+510	6 – 0	6 – 0	6 – 0	Premier League
fresco_e_fofo	18 – 00	+474	6 – 0	6 – 0	6 – 0	Premier League
nos_os_tres	18 – 00	+418	6 – 0	6 – 0	6 – 0	Premier League
Barbosa	17 – 01	+436	6 – 0	6 – 0	5 – 1	Premier League
Quim_Farda	17 – 01	+414	6 – 0	6 – 0	5 – 1	Premier League
Poison	17 – 01	+396	6 – 0	6 – 0	5 – 1	Premier League
Spulit	16 – 02	+438	6 – 0	6 – 0	4 – 2	Premier League
ACP	16 – 02	+402	6 – 0	6 – 0	4 – 2	Premier League
MaPeVa	15 – 03	+362	6 – 0	6 – 0	3 – 3	Premier League
amadeo	15 – 03	+332	6 – 0	4 – 2	5 – 1	Premier League
SrDaPedra	15 – 03	+290	6 – 0	6 – 0	3 – 3	Premier League
SimpleATaxx	15 – 03	+288	6 – 0	6 – 0	3 – 3	Premier League
casajo	15 – 03	+220	5 – 1	6 – 0	4 – 2	Premier League
Ghanima	15 – 03	+102	6 – 0	5 – 1	4 – 2	Premier League
CR_Ataxx	14 – 04	+284	6 – 0	6 – 0	2 – 4	League of Honor
ataxx_fighters	14 – 04	+250	6 – 0	6 – 0	2 – 4	League of Honor
MATEP	13 – 05	+298	6 – 0	5 – 1	2 – 4	League of Honor
Ataxx05	13 – 05	+254	6 – 0	5 – 1	2 – 4	League of Honor
GTG	13 – 05	+248	6 – 0	5 – 1	2 – 4	League of Honor
Nuno_Helder	13 – 05	+242	6 – 0	6 – 0	1 – 5	League of Honor
VA	13 – 05	+210	6 – 0	6 – 0	1 – 5	League of Honor
JPT	13 – 05	+118	6 – 0	5 – 1	2 – 4	League of Honor
MariJuAna	13 – 05	+118	6 – 0	5 – 1	2 – 4	League of Honor
nuno	12 – 06	+298	6 – 0	4 – 2	2 – 4	League of Honor
Nevermore	12 – 06	+248	6 – 0	4 – 2	2 – 4	League of Honor
Nos	12 – 06	+222	6 – 0	5 – 1	1 – 5	League of Honor
Trio_Marabilha	12 – 06	+102	6 – 0	4 – 2	2 – 4	League of Honor
jac	11 – 07	+238	6 – 0	3 – 3	2 – 4	League of Honor
DN	11 – 07	+166	6 – 0	5 – 1	0 – 6	League of Honor
afodanielus	11 – 07	+158	6 – 0	4 – 2	1 – 5	League of Honor
YAA	11 – 07	+154	6 – 0	5 – 1	0 – 6	League of Honor
BLACK_KNIGHT	10 – 08	+044	6 – 0	4 – 2	0 – 6	League of Honor
Khan	10 – 08	+036	6 – 0	4 – 2	0 – 6	League of Honor
Owl	09 – 09	+098	6 – 0	3 – 3	0 – 6	Unqualified
OMJIA05	08 – 10	+160	6 – 0	2 – 4	0 – 6	Unqualified
semGrupo	07 – 11	+070	6 – 0	1 – 5	0 – 6	Unqualified
GrimEater	07 – 11	–198	6 – 0	1 – 5	0 – 6	Unqualified
Mataxx	06 – 12	–94	5 – 1	1 – 5	0 – 6	Unqualified
BJM	06 – 12	–160	6 – 0	0 – 6	0 – 6	Unqualified
While1	06 – 12	–168	6 – 0	0 – 6	0 – 6	Unqualified
CVS	06 – 12	–170	4 – 2	2 – 4	0 – 6	Unqualified
MEandMe	02 – 16	–430	0 – 6	2 – 4	0 – 6	Unqualified
RPR	01 – 17	–484	1 – 5	0 – 6	0 – 6	Unqualified
SmallAtaxx	00 – 18	–882	0 – 6	0 – 6	0 – 6	Unqualified
tcx	00 – 18	–882	0 – 6	0 – 6	0 – 6	Unqualified

higher than 75% for programs in the Premier League, and between 60% and 75% for programs in the League of Honor. Unqualified programs are individually evaluated and are always graded lower than 60%.

In the Premier League, the best program developed by the teachers enters the competition and influences the ranking, which, in turn, influences the grades. The grades after the all-against-all competitions are determined as follows:

*Grade = 75 + (25 * Wins / Winner Wins)* for Premier League;

*Grade = 60 + (15 * Wins / Winner Wins)* for League of Honor.

Table 1 presents the results of the preliminary qualification round. We show the number of wins and losses against each opponent (W–L). We also included the *goal average* of each program, i.e. the total sum of its pieces minus the total sum of the opponent pieces. A total of 15 programs qualified for the Premier League, while 19 programs qualified for the League of Honor. The remaining 12 programs did not qualify, unable to have a higher number of wins on the confront against Random, Greedy and Level1.

We then proceeded to an all-against-all competition. Regarding the league competitions, we only present results for the highly competitive Premier League, where a program developed by the first author of this paper, named *Harkonnen*, joined the 15 qualified programs from the students. This competition involved a total of 480 matches. Table 2 summarizes the results of this competition. Complete results of this and other leagues, including loadable game logs can be seen on (Ribeiro *et al.*, 2008).

Harkonnen, our own program, was able not to lose a single game against the students programs, while playing as first player. However, as second player it lost 4 games, which

Table 2

Premier League results

#	Program	Total			1st Player			2nd Player			Grade
		Win	Loss	G. A.	Win	Loss	G. A.	Win	Loss	G. A.	
1	Harkonnen	56	04	+566	30	00	+328	26	04	+238	100.0
2	Toninja	50	10	+878	26	04	+352	24	06	+526	97.5
3	fresco_e_fofo	46	14	+436	22	08	+122	24	06	+314	95.5
4	genocide	46	14	+362	24	06	+128	22	08	+234	95.5
5	Quim_Farda	37	23	−72	16	14	+062	21	09	−134	91.5
6	Barbosa	36	24	+168	18	12	+064	18	12	+104	91.0
7	Poison	35	25	+594	15	15	+182	20	10	+412	90.5
8	nos_os_tres	29	31	−76	13	17	+002	16	14	−78	88.0
9	casajo	26	34	−366	17	13	−122	09	21	−244	86.5
10	ACP	25	35	+150	12	18	+188	13	17	−38	86.0
11	MaPeVa	25	35	−664	15	15	−222	10	20	−442	86.0
12	Spulit	21	39	+032	10	20	+084	11	19	−52	84.5
13	Ghanima	15	45	−264	07	23	−182	08	22	−82	81.5
14	SrDaPedra	15	45	−400	09	21	−136	06	24	−264	81.5
15	amadeo	12	48	−812	08	22	−420	04	26	−392	80.5
16	SimpleATaxx	06	54	−532	02	28	−344	04	26	−188	77.5

is quite revealing of the effort developed by students to produce their programs, given the level of sophistication put in the optimization of *Harkonnen*.

5. The Winning Programs

Since Ataxx is not a very well known game, there exists almost no literature on it. However, Ataxx is a zero-sum, perfect information game. Therefore, the classical and logical approach to it involves basically the well known minimax algorithm (Shannon, 1950), and its variations or optimizations, like the alpha-beta pruning (Knuth and Moore, 1975). We encouraged the students to use this approach as a starting point, and this strategy concepts were taught on the theorical classes. However there were no imposed boundaries on what they could use, besides the fact that they should create a pure Prolog program.

In this section we will start by describing the program that obtained the first place in the tournament and then proceed to reveal some of the most interesting ways the students used to attack the Ataxx problem.

5.1. *The Harkonnen Agent*

The agent's name, *Harkonnen*, was inspired by Frank Hebert's Dune, one of the most influential sci-fi novels of all time, and first published in 1965. The *Harkonnen* are a race known by its aggressive nature, always wanting to attack, a strategy that fitted the purpose of the agent.

The first approach was to apply the alpha-beta pruning (ABP). In this algorithm, the heuristic function is very important, since non ending positions must be evaluated and measured by a single number, that obviously could induce errors. Several heuristics functions were tried, but in the end the best behaviour was obtained with the most simple function $h(board) = number_of_our_pieces - number_of_their_pieces$. All other functions (which evaluated things like positional gain or stability) introduced a overhead in the computation (reducing the searchable depth) and did not provide a significative increase of accuracy.

Given this, the greatest limitation of the program lied precisely on the fact that it had to be programmed in Prolog, which does not have a comparable performance to more low-level languages like C (even using Yap). Note that Ataxx has a very large branching factor (the average is about 60) and that a typical game takes more than 70 moves. If we add the fact that almost all combinations of pieces in the board can be reached (remember the instability of a board position), the game complexity grows even further. In fact, the total number of different games that can be played is bigger than 10^{800}. Our approach to this fact was to apply several optimizations to the Prolog program.

The first one was to modify the internal representation of the board. Instead of using lists, a numerical representation was used. The board is divided in two parts (first four lines and last three lines) and each one is represented by two integers, where the first one indicates if the correspondent positions are empty or with a piece, and the second one represents the type of piece that is on that position. The numbers are binary encoded

(where 0/1 indicate an empty/filled square or our/opponent piece), and that is why we had to divide the board in two parts (to make the numbers fit in a normal native 32-bit integer). With this, we can use simple bit operations for our calculations, which are native to the processor. For example, knowing if a determined square is empty can be easily verified by a single bitwise conjunction. And swapping a piece can be made by a bitwise disjunction. This originated a huge boost on the program speed.

Besides keeping the four numbers that depict the board, the representation used also keeps the number of pieces in each side. With this, the heuristic function can be easily applied by doing a single subtraction. Maintaining the representation correct can be done by simply changing the past piece quantities, in order to reflect the move that was made and the actual squares that changed (not needing to do a fresh recount of all the pieces).

A single number representation was also used for each possible move, and equivalent moves were not evaluated. For example, if we add a new piece on a determined square, it does not matter if the original piece was its bottom or upper neighbour, because the board that it creates is the same. Hence, there are 529 possible moves, distributed by 49 "add" moves and 480 "jump" moves.

Given all this, the fact that Yap Prolog used indexing on the first predicate argument was exploited and specific predicates for each possible move were made. These hundreds of predicates were stored on an auxiliary file and they were automatically generated by another program.

Other small optimization was to pre-calculate the initial moves with a longer depth, and store them. In this way, the first two moves of *Harkonnen* were always instantaneous, and very accurate.

One problem was still open. How to cope with the time constraints? Remember that the program has only 4 minutes available for all the moves. The most basic solution is to introduce a limit in the depth used in the ABP. However, the time it takes to search to a specified depth is largely dependent on the board itself. Open positions have a larger branching factor. Besides, when we approach the end of a game, it is even more crucial to search to larger depths, since that can have a decisive impact on winning the game. To solve that problem, an iterative deepening strategy was used, that could dynamically adapt to each game. It can be seen that each new level of depth introduces a bigger magnitude of time in the search, that is, searching all the depths until N is only a small fraction of depth $N + 1$. *Harkonnen* uses a fixed maximum amount of time for each move, and tries to go as deep as it can go without surpassing the allotted time limit. More than that, the previous searches of smaller depths are used to sort the moves, in order to increase the probability of cutting tree branches with the ABP. *Harkonnen* also uses different time limits and different initial depths for the iterative deepening, based on the state of the game. These variables were manually tuned to better fit the constraints for this particular tournament, but its general design permits the agent to adapt to different circumstances.

Other possibilities for the agent were considered, like the negascout (Reinefeld, 1989) or MTD(f) (Plaat *et al.*, 1995) algorithms, as well as minor modifications such as aspiration windows, transposition tables, killer moves or quiescence search. However these were not applicable to Ataxx (for example, quiescence search does not make sense since

all board positions are chaotic) or they did not provide enough speed gains that would compensate its greater instability (like it was the case in using negascout). Furthermore, maintaining the simple alpha-beta pruning algorithm would add to the simplicity and elegance of the algorithm.

5.2. *Interesting Strategies*

Almost all the students opted to go in the direction we pointed, that is, they opted for the ABP, and they all ended in using the same simple heuristic function of Harkonnen. However, there were some interesting variations.

The first three student programs opted for creating a large number of specific predicates for things like moves or board access, using the first argument indexing property of Yap, similarly to Harkonnen. They typically generated clauses using another imperative program (for example in C). The *Toninja* agent used move ordering to guide the search, pre-calculation of some initial moves, an optimized search for possible moves and a formula based on the number of free squares and the available time to calculate to depth of the search. The *Fresco_e_Fofo* agent used negascout (with a reported increase of 20 % in its speed) and iterative deepening, and it also pre-calculates the initial moves. *Genocide* opted for the MTD(f) algorithm (they reported a 40% gain in its speed in relation to their initial ABP implementation) and a strategy comparable to iterative deepening.

The other agents were similar, and their main variation was in the way they managed time (some had a static depth, others had formulas similar to *Toninja*), the level of optimization and the correctness of their ABP implementation. Another interesting aspect is that almost all the programs were deterministic, giving the same moves for a determined board position. Only one agent (*Ghanima*) used randomness in its strategy, in such a strong way that playing against the same deterministic opponent it would sometimes wins and sometimes lose.

6. Feedback from Students

In order to better understand the students' reaction to our teaching approach, we promoted a survey, which was only made more than two years after the actual experience. We had 37 responses out of about 100 students that took the course and made an agent (we sent an email). We used the 5 point agree-disagree scale. Table 3 summarizes the responses obtained.

We can see that clearly our overall approach was liked by the students. The motivation factor with the most impact was using a game. The competitive factor was also deemed as very positive, although it was not unanimous. In what concerns to really using the results of the competition has the only grading system, we can see that students are more cautious, and a significative portion of them has a neutral stand on the subject. It should be noted however that given it is a completely objective evaluation, there was not a single student who disagreed with that method for grading. Generally, students also thought that

Table 3

Feedback from all students

Question	Strongly Agree	Agree	Neutral	Disagree	Strongly Disagree
Using a game as the development platform has motivated me	72.97%	24.32%	2.70%	0%	0%
The competitive aspect of the assignment has motivated me	56.76%	32.43%	5.41%	0%	5.41%
Evaluate using performance on the tournament is fair	25.00%	52.77%	22.22%	0%	0%
The assignment helped me to better understand Prolog	47.22%	47.22%	0%	2.77%	2.77%
This teaching strategy should be adopted in other courses	70.27%	18.92%	8.11%	0%	2.70%

Table 4

Feedback from students who were repeating the course

Question	Strongly Agree	Agree	Neutral	Disagree	Strongly Disagree
Compared to past years, I preferred this teaching strategy	60%	24%	12%	0%	4%
Compared to past years, I learned Prolog better	52%	32%	4%	8%	4%

the assignment was useful in learning Prolog and they were very assertive in stating that they would like something like this experience to be used on other courses.

We prepared also two more questions for the student who were repeating the course, which was the case for 25 of them, and the results are summarized in Table 4. The general opinion shows that our learning approach compares well to the more classical teaching paradigm.

We gave the students the opportunity to detail more their feedback by using custom messages, and we were pleasantly surprised to see that there were a lot of them who still had fond memories of this experience. Even with so much time in between, there was even the case of two students from different top teams that are now working on the same company and ended up discussing the assignment, because both liked it very much. Out of all responses, the competitive factor emerged has the strongest motivational aspect.

7. Conclusions

This paper described the work developed in the context of the undergraduate teaching of AI at DCC-FCUP. The main goal of the paper was to report on the impact that the

competitive setup caused on students' commitment, which surpassed our most optimistic expectations. The level of participation was outstanding and the depth of the AI concepts students employed showed that students went beyond what they learned in the classes, taking the initiative to use other sources to improve their knowledge on the subject. We also conducted a survey which shows that we were able to really increase the motivation, and in doing it effectively improving the learning experience.

Regarding the declarative nature of the LP contenders submitted by students, the results were also interesting. As said, the AI subject is taught on the third year of our Computer Science course. Hence, students have already an important amount of experience with lower-level programming languages, compilation technology and computer architecture, which the top contenders clearly explored to make their programs more efficient. As a result, the teaching of LP itself, its high-level and declarative nature bastions, cannot be evaluated by looking at the source Prolog code of the top contenders. This source code resembles much more assembly code than high-level and readable Prolog code, as top students have written programs in other languages, such as C, to produce the assembly-like Prolog code. This further emphasizes the level of commitment that the competition framework described in this paper was able to create in students, although it also shows that competition can be a pitfall in making the students produce declarative Prolog programs.

Concluding, this is certainly a teaching strategy that we would consider to use in the future. The positive points we identified surpass the negative ones and makes the competitive game an approach that provides a very nice learning environment both for the students and the teachers.

References

Ataxx in Wikipedia. `http://en.wikipedia.org/wiki/Ataxx`

Bergin, S. and Reilly, R. (2005). The influence of motivation and comfort-level on learning to program. In *Proceedings of the 17th Workshop on Psychology of Programming – PPIG'05*, 293–304.

Beyrand, A. (2005). *The Pressibus Inventory of Ataxx and Hexxagon Games*. `http://www.pressibus.org/ataxx/`

Costa, V. (2008). *YAP Prolog*. `http://www.dcc.fc.up.pt/ vsc/Yap/`

Filgueiras, M. (1987). Generating Natural Language sentences from semantic representations. In A. Sernadas and J.M. Neves (Eds.), *Actas do Terceiro Encontro Português de Inteligência Artificial*, 246–256.

Hingston, P., Combes, B. and Masek, M. (2006). Teaching an undergraduate ai course with games and simulation. In *Edutainment*, 494–506.

Knuth, D.E. and Moore, R.W. (1975). An analysis of alpha-beta pruning. *Artificial Intelligence*, **6**(4), 293–326.

Lawrence, R. (2004). Teaching data structures using competitive games. *IEEE Transactions on Education*, **47**(4), 459–466.

Leal, J.P. and Silva F. (2003). Mooshak: a Web-based multi-site programming contest system. *Software*: *Practice and Experience*, **33**(6), 567–581.

Martins, J.P. and Morgado, E.M. (Eds.) (1989). *EPIA 89, 4th Portuguese Conference on Artificial Intelligence, Lisbon, Portugal, September 26-29, 1989, Proceedings*. *Lecture Notes in Computer Science*, Vol. 390. Springer. `http://dblp.uni-trier.de`

Plaat, A., Schaeffer, J., Pijls, W. and de Bruin, A. (1995). *A New Paradigm for Minimax Search*. EUR-CS-95-03, Rotterdam, Netherlands.

Porto, A. and Filgueiras, M. (1984). Natural language semantics: A logic programming approach. In *SLP*, 228–232. `http://dblp.uni-trier.de`

Project Swing. http://java.sun.com/j2se/1.5.0/docs/guide/swing/
Reinefeld, A. (1989). Spielbaum-Suchverfahren. *Informatik-Fachberichte*, **200**.
Ribeiro, P., Simões, H. and Ferreira, M. (2008). Site with auxiliar material for the paper. http://www.dcc.fc.up.pt/~pribeiro/ataxx/
Roy, P.V. (1994). 1983–1993: The wonder years of sequential Prolog implementation. *Journal of Logic Programming*, **19**(20), 385–441.
Shannon, C.E. (1950). Programming a computer for playing chess. *Philosophical Magazine*, **41**, 256–275.
Tomás, A.P. and Filgueiras, M. (1989). Some comments on a logic programming approach to natural language semantics. In *EPIA*, 187–197. http://dblp.uni-trier.de
Wallace, S.A. and Margolis, J. (2007). Exploring the use of competetive programming: observations from the classroom. *J. Comput. Small Coll.*, **23**(2), 33–39.

P. Ribeiro is currently a PhD student at the University of Porto, where he completed his computer science degree with top marks. He has been involved in programming contests (PC) since a very young age, representing its school and country in several national and international PC. He now actively participates in the organization of PC, creating and discussing problems, being responsible for the training campus of the portuguese team in the International Olympiad in Informatics (IOI) and since 2005 also serving as deputy leader for the portuguese delegation in IOI. He is currently researching parallel algorithms for pattern mining in complex networks.

H. Simões is a computer scientist graduate of the University of Porto. Currently, he is working towards his PhD degree on compile-time analyses for resource usage prediction of functional programs. He is a researcher in LIACC (Artificial Intelligence and Computer Science Laboratory), Portugal.

M. Ferreira is currently an assistant professor at the Computer Science Department of the School of Sciences of University of Porto and director of the under-graduate course in computer science, where he has been teaching courses related to artificial intelligence, logic programming and advanced databases. His research interests evolved from the implementation of logic programming systems, where he has been contributing on the development team of the Yap Prolog System (the MYDDAS deductive database module), to the area of logic-based spatial databases and spatial networks. He has been leading several research projects in the areas of deductive databases, vehicular networks and advanced spatial database systems.

Dirbtinio intelekto ir loginio programavimo mokymas konkurencingoje aplinkoje

Pedro RIBEIRO, Hugo SIMÕES, Michel FERREIRA

Motyvacija vaidina svarbų vaidmenį mokymo procese. Šis straipsnis supažindina su Porto universiteto Gamtos mokslų fakulteto Informatikos katedros patirtimi mokant baigiamojo kurso studentus dirbtinio intelekto. Buvo sukurta sudėtingos konkurencijos darbo sistema, apimanti “Prolog” kalba suprogramuotą konkurentą, žaidimų serverį bei grafinę sąsają, kuri skatintų studentus įsigilinti į pateiktas užduotis. Taip pat pateikiamas tokios konkurencijos poveikis studentų atsidavimui, kuris atitiko pačius optimistiškiausius tyrėjų lūkesčius.