CIS335: Logic Programming, Assignment 5 (Assessed)

Game Playing in Prolog

Geraint A. Wiggins

November 11, 2004

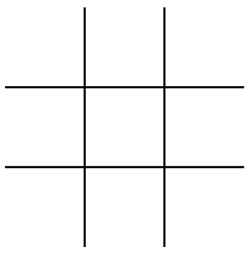
1 Introduction

This assignment is the last formally assessed course work exercise for students on the CIS335 module. The intent is to implement a simple game – Noughts and Crosses – from first principles. Certain parts of the program (such as input/output) are supplied in library files.

This practical counts for 13% of the marks for this course. In itself, it is marked out of 100, and the percentage points available are shown at each section.

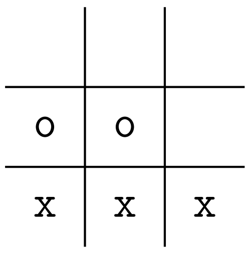
2 How to play the game

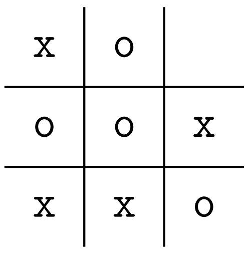
The rules of Noughts and Crosses are simple. We have a board which is 3×3 squares large, like this:



We will number the squares from left to right and top to bottom, so the top left is $(1, 1)$, the bottom left is $(1, 3)$ and the bottom right is $(3, 3)$.

There are two players, `o` and `x`, and they take turns in occupying successive squares on the board until someone wins or the board is full or stalemate is reached. One one player may occupy a square, and once taken a square stays taken. The winner of the game, if there is one, is the player who gets a complete line of pieces, across, down, or diagonally. Here are two examples, one of a win for `x` and one of a stalemate, where no-one can win:





There are some simple strategies which will help a computer win at Noughts and Crosses. They do not involve any real planning, but can often lead to a win, or at least hold off a loss. Applied in this order, they are:

1. If there is a winning line for self, then take it;
2. If there is a winning line for opponent, then block it;
3. If the middle space is free, then take it;
4. If there is a corner space free, then take it;
5. Otherwise, dumbly choose the next available space.

We will use these simple *heuristics* at the end of the practical.

3 The Implementation

3.1 Program structure

This practical is quite strictly structured, so as to give a feel for how good program design is done. Even if you are an experienced programmer, please follow the style here. In particular, you *must* follow the instructions for data-representation, or else the supplied code will not work.

3.2 Library software

In this practical, you will be using libraries, `io.pl` and `fill.pl`, and probably the built-in `lists.pl`. You access the libraries by using the `use_module/1` predicate – just download `io.pl` and `fill.pl` from the web site, store them in the same directory as the rest of your work for this assignment, and put the following at the top of your program file:

```
:-  use_module(  [library(lists),
                  io ),
                  fill )] ).
```

The three library modules contain useful predicates, which saves you repeating other people's work. The `lists` library is documented in the SWIProlog manual.

The other two libraries are built specifically for this practical. `io.pl` exports seven predicates, which work as follows:

`display_board/1` prints out a representation of the board, depending on what symbols you have used to represent noughts, crosses and blank spaces. Its argument is your representation of the board. If the representation is correct, it always succeeds. Argument: `Board`.

`get_legal_move/4` requests the coordinates of a board square, checks that it is empty, and returns the coordinates to the main program. It keeps asking until legal coordinates are given (*ie* a square on the board which is not already taken). If the representation is correct, it always succeeds. Arguments: `Player, X-coordinate, Y-coordinate, Board`.

`report_move/3` prints out a move just selected. If the representation of the board is correct, it always succeeds. Arguments: `Player, X-coordinate, Y-coordinate`.

`report_stalemate/0` prints out a warning that there is a stalemate and the game is drawn. It always succeeds.

`report_winner/1` prints out a warning that there is a winner – the player named in the one argument. It always succeeds. Argument: `Player`.

`welcome/0` prints out a welcome to the game. It always succeeds.

`fill.pl` contains one predicate:

`fill_square/5` which takes a coordinate pair, a player, and a board representation, and replaces the square indicated by the coordinates with the piece for the player, to give a new board representation. Arguments: `X-coordinate, Y-coordinate, Player, Old-board, New-board`. It succeeds if the input representations are correct, and does not check for illegal moves.

You can look at the definitions of these predicates in the files `fill.pl` and `io.pl` if you want to, but you do not need to do so to complete this practical.

3.3 The Practical

The following sections lead you through the practical step by step. You should be able to test your code at all times, and you will not need anything beyond what has been covered in the lectures or what is in the libraries. You do not need to understand how the library code works to complete the practical. It is *imperative* that you follow the instructions closely; otherwise, some of the library code, which uses your code, may not work.

3.4 Board representation (15%)

Design a representation for the board, using some kind of term representation. You will need a one-character symbol for each player and another for a blank space on the board. (Each needs to be one-character to fit in with the library software.)

Implement the following predicates. (Don't worry about error checking in your program – just make sure predicates succeed when you want them to, and fail at all other times.) Wherever possible, implement each predicate in terms of predicates you have already defined. Note that you may not need to use all these predicates in your final program, but some of them are used in the libraries. All of these predicates may be called in any mode – that is, you should not assume that any argument will be instantiated.

`is_cross/1` succeeds when its argument unifies with the cross character in your representation.

`is_nought/1` succeeds when its argument unifies with the nought character in the representation.

`is_empty/1` succeeds when its argument unifies with the empty square character in the representation.

`is_piece/1` succeeds when its argument unifies with either the cross character or the nought character.

`other_player/2` succeeds when each of its arguments is a different player representation characters.

`row/3` succeeds when its first argument is a row number (between 1 and 3) and its second is a representation of a board state. The third argument will then be a term like this: `row( N, A, B, C )`, where N is the row number, and `A, B, C` are the values of the squares in that row.

`column/3` succeeds when its first argument is a column number (between 1 and 3) and its second is a representation of a board state. The third argument will then be a term like this: `col( N, A, B, C )`, where N is the column number, and `A, B, C` are the values of the squares in that column.

`diagonal/3` succeeds when its first argument is either `top_to_bottom` or `bottom_to_top` and its second is a representation of a board state. The third argument will then be a term like this: `dia( D, A, B, C )`, where `D` is the direction of the line (as above), and `A, B, C` are the values of the squares in that diagonal. The diagonal direction (*eg* top-to-bottom) is moving from left to right.

`square/4` succeeds when its first two arguments are numbers between 1 and 3, and its third is a representation of a board state. The fourth argument will then be a term like this: `squ( X, Y, Piece )`, where (X,Y) are the coordinates of the square given in the first two arguments, and Piece is one of the three square representation characters, indicating what if anything occupies the relevant square.

`empty_square/3` succeeds when its first two arguments are coordinates on the board, and the square they name is empty.

`initial_board/1` succeeds when its argument represents the initial state of the board.

`empty_board/1` succeeds when its argument represents an uninstantiated board (*ie* with Prolog variables in all the spaces). *Don't confuse this with a board all of whose squares contain the "empty" character!*

3.5 Spotting a winner (15%)

We need a predicate to tell us when someone has won.

`and_the_winner_is/2` succeeds when its first argument represents a board, and the second is a player who has won on that board. (Hint: use the predicates above here; you need 3 clauses.)

Test your predicate on some hand-made data.

3.6 Running a game for 2 human players (20%)

To start off with, we will build a program which acts as a board for two human players, displaying each move, and checking for a win or draw.

We will assume that `x` is always going to start. We will use a predicate called `play/0` to begin a game, defined as follows:

```
play  :-  welcome,
          initial_board( Board ),
          display_board( Board ),
          is_cross( Cross ),
          play( Cross, Board ).
```

You will need to define two predicates to make this work:

`no_more_free_squares/1` succeeds if the board represented in its argument has no empty squares in it.

`play/2` is recursive. It has two arguments: a player, the first, and a board state, the second. For this section of the practical, it has three possibilities:

1. The board represents a winning state, and we have to report the winner. Then we are finished.
2. There are no more free squares on the board, and we have to report a stalemate. Again, we are finished.
3. We can get a (legal) move from the player named in argument 1, fill the square he or she gives (using the `fill.pl` library), switch players, display the board and then play again, with the updated board and the new player.

Don't forget that many of the things you need to do this are defined for you in the library!

When you get to this point, test out your program thoroughly, playing several games, trying out the various possibilities for winning, drawing *etc.*

4 Running a game for 1 human and the computer (20%)

Having checked out the part of the program which runs the game and displays it, we now need to extend the program to play itself. We will assume that it will always play `o`.

To do this, we will need to adapt step 3 of the `play/2` predicate defined above. Place that part of your code in `/*...*/` to comment it out, and put the text "`code for section 3.6`" in the comment.

We have to replace the third part of `play/2` with two new parts:

`play/2` *contd.* The second version of `play/2` has two possibilities:

3. The current player is `x`, we can get a (legal) move, fill the square, display the board, and play again, with the new board and with nought as player (this is very like part 3 above).
4. The current player is `o`, we can choose a move (see below), we tell the user what move we've made (see `io.pl` library), we can fill the square, display the board, and play again, with the new board and with cross as player.

5 Implementing the heuristics (20%)

In order to make the computer play, we need to implement one more predicate:

`choose_move/4` which succeeds when it can find a move for the player named in its first argument, at the (X,Y)-coordinates given in its second and third arguments, respectively. It has five alternatives, corresponding with the heuristics given in section 2; the last clause looks like this:

```
% dumbly choose the next space
choose_move( _Player, X, Y, Board )  :-  empty_square( X, Y, Board ).
```

5.1 Spotting a stalemate (10%)

Finally, it would be nice to spot a stalemate as soon as it happens, so that the players don't have to go on until the whole board is full. To do this, you will need to replace step 2 of `play/2` as follows. When you do this, comment out the code for the old step 2 and add the text "`code for section 3.6`" in the comment, as above.

`possible_win/2` is recursive. It succeeds if the addition of one square to the board (represented in the second argument) yields a win for a player (represented in the first argument). Alternatively, it succeeds if the result of adding one square to the board leads to a possible win, swapping players as it goes. In total, it succeeds if any player can win from the current position, allowing for whose move it is now.

`play/2` *contd.* We replace the second step of `play/2` above as follows:

2. If no possible win is available, report a stalemate. Then we have finished.

6 Assessment

The minimal core of this practical is considered to be the part up to and including section 3.6; most students should expect to finish the whole program.

7 Submission

By 12 noon on Friday 10th December, you must submit your solution as an email attachement, called `cis335-YOURLOGIN-assignment5.pl` (where `YOURLOGIN` is your College login name) to `g.wiggins@gold.ac.uk`. You must send the email from your College account; otherwise it will not be accepted. This should consist of the documented prolog code, including the commented out parts described above in a *single loadable file.* You will be penalised if your program will not load when the marker tries it.