CIS335 2008-9, Practical 5 (Assessed)

Game Playing in Prolog

Geraint Wiggins

November 19, 2008

1 Introduction

This practical is the second formally assessed exercise for students on the CIS335 module. The intent is to implement a simple game – Othello (also known as Reversi) – from first principles. Certain parts of the program (such as input/output) are supplied in library files.

This practical counts for 13.3% of the marks for this module. 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 Othello are simple. We have a board which is 8×8 squares large, like a chess board. We will number the squares from left to right and top to bottom, so the top left is $(1, 1)$, the bottom left is $(1, 8)$ and the bottom right is $(8, 8)$.

There are two players, black ($\star$) and white ($\circ$), and they take turns in occupying successive squares on the board until the board is full or neither player can move. Only one player may occupy a square.

A piece may only be placed on the board if, in doing so, the player encloses one or more opponent's pieces between the new piece and an existing one of his or her own, horizontally, vertically, or diagonally, in a straight line. When this happens, the enclosed pieces change colour to match the enclosing ones. The winner of the game, if there is one, is the player with the most pieces on the board when it is full or noone can move further.

Initially, the board is laid out like this:

			$\circ$	$\star$			
			$\star$	$\circ$			

There are some simple strategies which will help a computer win at Othello. 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 it is possible to take a corner square, do so;
2. If it is possible to block the opponent from taking a corner square, do so;
3. If it is possible to take an edge square, do so;
4. If it is possible to block the opponent from taking an edge square, do so;
5. Otherwise, choose the available space which converts most of the opponent's pieces into your own.

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` and `fill`, and probably `lists`. You can download the libraries from the course home page, and then load them into Prolog by using the `use_module/1` predicate: just 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 SWI Prolog manual; you may find the `nth1/3` predicate particularly useful.

The other two libraries are built specifically for this practical. `io` 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 sensible coordinates are given – that is, of a square which is not taken, but which, if taken, will convert the opponent's pieces. This predicate calls `enclosing_piece/7`, which you are required to define. 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_no_move/1` prints out a warning that the player whose piece is given in the argument cannot move legally. It always succeeds.

`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.

The library `fill` exports one predicate:

`fill_and_flip_squares/5` succeeds if its last argument, a board representation, is the result of placing a piece at position (`X`,`Y`) on the board representation in its fourth argument. The coordinates are the first and second arguments, and the kind of piece to be placed is given in the third. It succeeds if the input representations are correct, and does not check for illegal moves.

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 (20%)

Design a representation for the board, using some kind of term representation. You will need a one-character symbol for each player and one for a blank space on the board. (It needs to be one-character to fit in with the library software.) If you want to use characters other than letters, you can do so by enclosing them in single quotes, such as `'*'`.

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 when the predicate is called.

`is_black/1` succeeds when its argument is the character representing a black piece in the representation.

`is_white/1` succeeds when its argument is the character representing a white piece in the representation.

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

`is_piece/1` succeeds when its argument is either the black character or the white character.

`other_player/2` succeeds when both its arguments are player representation characters, but they are different.

`row/3` succeeds when its first argument is a row number (between 1 and 8) 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, D, E, F, G, H )`, where `N` is the row number, and `A, B, ..., H` are the values of the squares in that row.

`column/3` succeeds when its first argument is a column number (between 1 and 8) 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, D, E, F, G, H )`, where `N` is the column number, and `A, ..., H` are the values of the squares in that column.

`square/4` succeeds when its first two arguments are numbers between 1 and 8, 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 (which is specified in argument 3), and the square they name is empty.

`initial_board/1` succeeds when its argument represents the initial state of the board, as shown in Section 2.

`variable_board/1` succeeds when its argument unifies with a representation of the board with distinct variables in the places where the pieces would normally go.

3.5 Spotting a winner (20%)

We need a predicate to tell us when someone has won. To do this, we need to count the pieces and empty squares, and compare them. Construct the following predicates:

`count_pieces/3` succeeds when its first argument is a board representation and its second and third arguments are the number of black and white pieces, respectively. One way to structure this predicate is to write an auxiliary predicate which deals with just rows or just columns (whichever is more natural for your representation) first.

`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. It can be defined using `count_pieces/3`, and in that case will probably have two clauses.

Test your predicates 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 black 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_black( Black ),
          play( Black, Board ).
```

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

`enclosing_piece/7` has arguments `X`, `Y`, `Player`, `Board`, `U`, `V`, `N`. It succeeds if a piece of type `Player` placed at (`X`,`Y`) would enclose `N` opponent's pieces between itself and the piece belonging to `Player` at (`U`,`V`).

`no_more_legal_squares/1` succeeds if the board represented in its argument has no more squares in it onto which a legal move can be made. You may want to use `enclosing_piece/7` for this.

`no_more_legal_squares/2` succeeds if the board represented in its second argument has no more squares in it onto which a legal move can be made by the player whose piece is given in the first. You may want to use `enclosing_piece/7` for this.

`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. No more moves are possible, and we have to report the winner. Then we are finished.
2. No more moves are possible, but there is a draw. Then we are finished.
3. No legal move is available for the current player (whose piece is represented in argument 1), in which case we play again, this time with the opponent.
4. We can get a (legal) move from the player whose piece is given in argument 1, fill the square he or she gives, convert any opponent's pieces as appropriate, switch players, display the board and then play again, with the updated board and the new player.

When you get to this point, test out your program thoroughly, playing several games, trying out the various possibilities for winning, drawing *etc*. (Hint: you don't need to play a whole game to test each feature – you can set up dummy `initial_board/1` predicates to test particular situations easily.)

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 for itself. We will assume that it will always play `o`.

To do this, we will need to adapt step 4 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 step 4 of `play/2` with two new parts:

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

4a. The current player is `*`, 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 4 above).

4b. The current player is `o`, we can choose a move (see below), we tell the user what move we've made (see `msc/othello/io` 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. As a hint, a clause to choose the first legal space it finds would be like this:

```
% dumbly choose the next space
choose_move( Player, X, Y, Board )  :-
     empty_square( X, Y, Board ),
     enclosing_piece( X, Y, Player, Board, _, _, _ )
```

but of course, your code will be more intelligent, and use the heuristics given above. If you wish, you may extend your code to be more specialised, but you *must* comment out such extras in the submitted code.

Two predicates which you may find useful in designing the heuristic part are `findall/3` and `keysort/2`. They are explained in the SWI Prolog manual, which you can find on line. You can see `findall/3` in action in the `fill` library. We covered it in the lectures on meta-programming.

6 Submission

By 7pm on Friday of week 11 (*i.e.,* the last day of term), you must place your solution in the submission directory supplied in your home file space. This should consist of the commented 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.