CIS335 Assignment 5 (Assessed)

Game Playing in Prolog

Geraint Wiggins

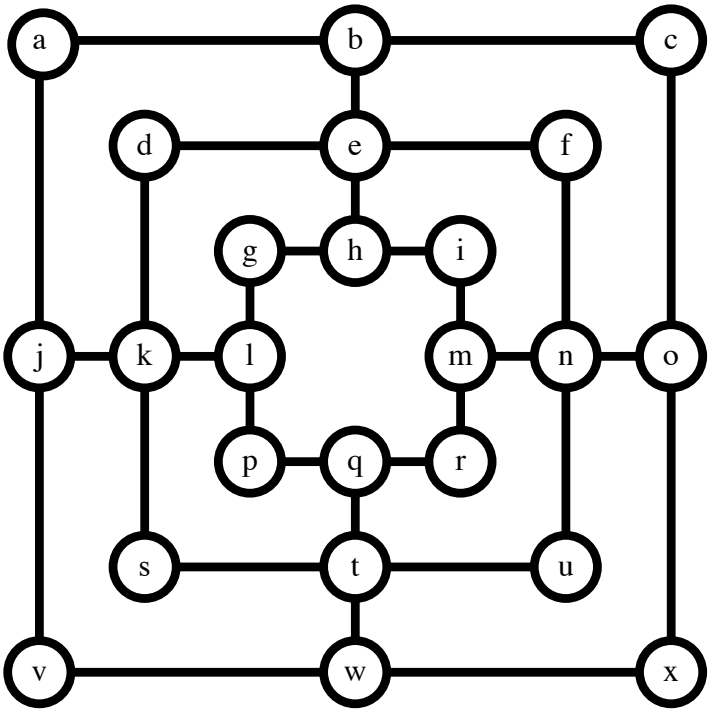
1 Introduction

This practical is the second formally assessed exercise for 3rd years students on the CIS335: Logic Programming module. The intent is to implement a simple game – Merels – from first principles. Certain parts of the program (such as input/output) are supplied in library files.

This practical counts for 10% 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 modern rules of Merels date from the Middle Ages, though there is evidence that a simpler version of the game existed as long ago as 1400BC. Like many games, the rules are simple, but the emerging possibilities are complex. We start with a board made up of lines connecting points, like this:



I have labelled the points with letters for easy reference. Note that you must stick to this labelling, otherwise the supplied library code may not work.

There are the rules of Merels, taken from Past Times' compendium of "Cloister Games":

> "Starting with nine men (merels) each, you place alternately one at a time on to any vacant point on the board. Each time you manage to form a [straight] row of three merels,

creating a *mill*, you may remove any one of your opponent's merels, though not one which is in a mill. When all the merels have been entered on to the board, you continue taking turns by moving one merel to an adjacent point along a line, with the object of making a mill. You win the game if you manage to block your opponent's merels so they cannot be moved, or if you reduce him/her to only two merels.

You may make or break the same mill any number of times, capturing one of your opponents merels each time you make a mill. If you are left with only three merels on the board forming a mill and it is your turn, you must [still] move and break the mill."

There are some simple strategies which will help a computer win at Merels. 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 mill to be made, make it; if opponent is able to make a mill, remove one of the relevant pieces;
2. If opponent is about to make a mill, block it if possible;
3. Place pieces on points with many connections, where possible;
4. Otherwise, move your pieces closer together.

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 the `lists` library and the `merels_support.pl` file which you can download from the CIS335 home page. You access the libraries by using the `use_module/1` predicate – just put the following at the top of your program file:

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

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

`http://gollem.science.uva.nl/SWI-Prolog/Manual/`

The other library is built specifically for this practical. `merels_support.pl` exports 11 predicates, which work as follows:

`display_board/1` prints out a representation of the board, depending on what symbols you have used to represent the two player's merels. Its argument is your representation of where the merels are on the board. If the representation is correct, it always succeeds. Argument: `Board`.

`get_legal_place/3` requests the name of a point, checks that it is empty, and returns it to the main program. It keeps asking until a legal name is given (*ie* a point on the board which is not already taken). If the representation is correct, it always succeeds. Arguments: `Player, Point, Board`.

`get_legal_move/4` requests the name of a point, checks that it is occupied by the current player, requests the name of another point, checks that it is connected and empty, and then returns both to the main program. It keeps asking until a legal point given. If the representation is correct, it always succeeds. Arguments: `Player, OldPoint, NewPoint, Board`.

`get_remove_point/3` requests the name of a point, checks that it is occupied by the opponent, and deletes the occupying merel. It keeps asking until legal names is given. If the representation is correct, it always succeeds. Arguments: `Player, Point, Board`.

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

`report_move/2` always succeeds and prints out a statement of a new placment. Argument 1 is the player, 2 is the point taken.

`report_move/3` always succeeds and prints out a statement of a new move. Argument 1 is the player, 2 is the point vacated, 3 is the point taken.

`report_remove/2` always succeeds. Argument 1 is the player, and Argument 2 is the point from which a merel is being removed.

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

`empty_point/2` succeeds if the point name which is its first argument is empty in the representation of the board which is its second. Beware of floundering here!

`check_mills/4` compares two consecutive board states and succeeds if a Player has made a new mill on the second. If appropriate it allows a Player to choose a merel to remove. It returns the final state of the board. Argument 1 is the Player, Argument 2 is the old board, Argument 3 is the new board, and Argument 4

You can look at the definitions of these predicates in the file `merels_support.pl`, 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 empty board, using a Prolog predicate. The representation should reflect what is important about the board for the purposes of the game, and it should not include superfluous information.

You will also need a one-character symbol for each player. (It needs to be one-character to fit in with the library software.) We will use a list of `point-merel` pairs to indicate which points on the board are taken by which player.

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. Unless otherwise stated, all of these predicates may be called in any mode – that is, you should not assume that any argument will be instantiated.

`is_player1/1` succeeds when its argument is the player 1 character in the representation.

`is_player2/1` succeeds when its argument is the player 2 character in the representation.

`is_merel/1` succeeds when its argument is either of the player characters.

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

`pair/3` succeeds when its first argument is a pair (in whatever representation you choose) made up of its second, a point, and its third, a merel.

`merel_on_board/2` succeeds when its first argument is a merel/point pair and is second is a representation of the merel positions on the board. Argument 2 is assumed to be instantiated.

`row/3` succeeds when its three arguments are (in order) a connected row, vertical or horizontal, in the board.

`connected/2` succeeds when its two arguments are the names of points on the board which are connected by a line (*i.e.,* there is a valid move between them).

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

Remember to document your chosen representation clearly in comments.

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 2 clauses, one for each way of losing.)

Test your predicate on some hand-made data.

3.6 Running a game for 2 human players (25%)

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.

We will assume that player 1 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_player1( Player ),
          play( 18, Player, Board ).
```

You will need to define a further predicate to make this work:

`play/3` is recursive. It has three arguments: the number of merels not yet placed, a player, and a board state. For this section of the practical, it has three possibilities:

1. All the merels have been placed, the board represents a winning state, and we have to report the winner. Then we are finished.
2. Not all the merels have been placed. We can get a (legal) placing from the player named in argument 1, fill the point he or she gives, check for any new mills, and ask which piece to remove if so, display the board, switch players and then play again, with the updated board and the new player.
3. All the merels have been placed. We can get a (legal) move from the player named in argument 1, move the merel he or she gives, check for any new mills, and ask which piece to remove if so, display the board, switch players 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*. A good thing to test is the working of the program with only 6 merels instead of the full 18.

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 be player 2.

To do this, we will need to adapt steps 2 and 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 parts 2 and 3 of `play/3` with two new parts each:

`play/3` *contd.* Each non-winning part of the second version of `play/3` has two possibilities:

a. Player 1 is current, we can get a (legal) move/placement, fill the square, display the board, and play again, with the new board and with player 2 as current player (this is almost exactly like the original `play/3` above).

b. Player 2 is current, we can choose a move/placement (see below), we tell the user what move or placement we've made (see `merels_support.pl`), we can fill the square, display the board, and play again, with the new board and with Player 1 current.

and you will also need a new version of `check_mill/4` - one which will choose which merel to remove by itself.

5 Implementing heuristics (25%)

In order to make the computer play, we need to implement three more similar predicates:

`choose_place/3` which succeeds when it can find a place to put a new merel. It will have several alternatives, corresponding with the heuristics you choose to implement; the last clause looks like this:

```
% dumbly choose a point
choose_place( _Player, Point, Board ) :-
                              connected( Point, _ ),
                              empty_point( Point, Board ).
```

`choose_move/4` which succeeds when it can find a merel to move and a place to move it to. It will have several alternatives, corresponding with the heuristics you choose to implement; the last clause looks like this:

```
% dumbly choose a move
choose_move( Player, OldPoint, NewPoint, Board ) :-
                             pair( Pair, OldPoint, Player ),
                             merel_on_board( Pair, Board ),
                             connected( OldPoint, NewPoint ),
                             empty_point( NewPoint, Board ).
```

`choose_remove/4` which succeeds when it can find a merel to remove. It will have several alternatives, corresponding with the heuristics you choose to implement; the last clause looks like this:

```
% dumbly choose a removal
choose_remove( Player, Point, Board ) :-
                                pair( Pair, Point, Player ),
                                merel_on_board( Pair, Board ).
```

You are invited to use the heuristics specified earlier and/or ones of your own invention to complete this practical. Note that you will NOT be marked on how well your program plays Merels – rather, assessment will be in terms of how well your program is implemented. Feel free to discuss Merels strategy with your friends, if you don't feel confident to make up your own heuristics, but want to use something better than those specified here. As usual, however, you MUST NOT share more than ideas: all the code you submit must be your own.

If you decide to implement your own heuristics, you MUST comment them clearly – otherwise, your cleverness may be confused with a mistake!

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, with one or two heuristics implemented.

7 Submission

By 7pm on Friday 4th December, you must submit your solution in electronic form, in the CIS335/5 assignment folder in your Department of Computing home space.