Jeson Mor - SICStus Prolog implementation

For detailed information about the predicates, please refer to the documentation.

Group Information

Jeson Mor_1

- Bruno Mendes - up201906166
- José Costa - up201907216

Contributions

Name	Contribution
Bruno Mendes	50%
José Costa	50%

Motivation

This project aims to provide an implementation of the chess variant `Jeson Mor` using the SICStus Prolog language distribution.

Install and execution

Apart from the standard SICStus Prolog installation, you should run the consult command on the `main.pl` file to load the game predicates. To run the game simply execute the `play` predicate.

Load the predicates:

```
consult('main.pl').
```

Run the game:

```
play.
```

Game description

The Jeson Mor chess variant, also known as `Nine Horses` is typically played in a 9x9 checkboard. Each piece is, as implied by the name, a horse, moving like the chess knight.

The goal of this Mongolia originated game is to move one horse to the middle cell and then move it out to any other cell. A player can also win by capturing all the opponent's horses.

This information was gathered in the `boardgamegeek` website, available in this location.

Implementation

Internal game representation

At each point, the game is represented by a state, a list with 3 elements [`CurrentPlayer`, `CurrentBoard`, `OldBoard`]. This list is refered throughout the code and documentation as `GameState`.

The `CurrentPlayer` element is an atom comprising either `w` or `b`, identifing the player turn.

The `CurrentBoard` element is a bi-demensional list containing the game board and pieces position's. The board rows are saved as elements of the outer list and the columns as elements of each row list. Column lists are saved from top to bottom and row lists left to right, just as they are printed. The pieces inside the board are represented by either `w` or `b` atoms, while empty cells are represented by an `o` atom.

The `OldBoard` element saves the board in the last turn, being useful for determining the game ending.

Below are some examples of `GameStates` in various situations:

Prolog `GameState` representations:

Initial state:

```
[w,
    [
        [b,b,b,b,b,b,b,b,b],
        [o,o,o,o,o,o,o,o,o],
        [o,o,o,o,o,o,o,o,o],
        [o,o,o,o,o,o,o,o,o],
        [o,o,o,o,o,o,o,o,o],
        [o,o,o,o,o,o,o,o,o],
        [o,o,o,o,o,o,o,o,o],
        [o,o,o,o,o,o,o,o,o],
        [w,w,w,w,w,w,w,w,w]
    ],
    [
        [b,b,b,b,b,b,b,b,b],
        [o,o,o,o,o,o,o,o,o],
        [o,o,o,o,o,o,o,o,o],
        [o,o,o,o,o,o,o,o,o],
        [o,o,o,o,o,o,o,o,o],
        [o,o,o,o,o,o,o,o,o],
        [o,o,o,o,o,o,o,o,o],
        [o,o,o,o,o,o,o,o,o],
        [w,w,w,w,w,w,w,w,w]
```

```
    ]
]
```

Mid game:

```
[b,
    [
        [b,o,o,o,o,o,b,o,b],
        [o,b,o,b,o,b,o,o,o],
        [o,o,o,b,o,b,o,o,o],
        [o,o,b,o,o,o,o,o,o],
        [o,o,o,o,w,o,o,o,o],
        [o,o,o,o,o,o,o,o,o],
        [o,o,o,o,o,o,o,o,o],
        [w,o,o,o,o,o,w,o,w],
        [w,w,o,w,w,o,o,w,o]
    ],
    [
        [b,o,o,o,o,o,b,o,b],
        [o,b,o,b,o,b,o,o,o],
        [o,o,o,b,o,b,o,o,o],
        [o,o,b,o,o,o,o,o,o],
        [o,o,o,o,o,o,o,o,o],
        [o,o,o,o,o,o,o,o,o],
        [o,o,o,o,o,w,o,o,o],
        [w,o,o,o,o,o,w,o,w],
        [w,w,o,w,w,o,o,w,o]
       ]
]
```

Final state:

```
[w,
    [
        [b,o,o,o,o,o,b,o,b],
        [o,b,o,b,o,b,o,o,o],
        [o,o,o,b,o,b,o,o,o],
        [o,o,b,o,o,o,o,o,o],
        [o,o,o,o,o,o,o,o,o],
        [o,o,o,o,o,o,o,o,o],
        [o,o,o,o,w,o,o,o,o],
        [w,o,o,o,o,o,o,o,w],
        [w,w,o,w,w,o,o,w,o]
    ],
    [
        [b,o,o,o,o,o,b,o,b],
```

```
        [o,b,o,b,o,b,o,o,o],
        [o,o,o,o,o,b,o,o,o],
        [o,o,b,o,o,o,o,o,o],
        [o,o,o,o,b,o,o,o,o],
        [o,o,o,o,o,o,o,o,o],
        [o,o,o,o,w,o,o,o,o],
        [w,o,o,o,o,o,o,o,w],
        [w,w,o,w,w,o,o,w,o]
    ]
]
```

Game display

User interaction happens in 3 cases. First for gathering game settings, then to interact with the board and lastly the game ending.

Input

Input happens in the first 2 cases, in game the settings, before the game loop, and then in the game itself.

Input is achieved by using predicates with this structure:

```
some_predicate:-
    repeat,
    display_prompt,
    read(Something),
    validate(Something)
```

When asking for atom representations this verification is made to ensure the input is treated as such:

```
    catch(char_code(_, Something), _, fail),
```

The user must finish all inputs with `.` (a dot) so that it gets parsed.

Movement parser Moves are read from input using algebraic notation, as in chess (start square to end square; eg. a1-b3), and then parsed to an internal move representation, where the column and row are treated to match the board's list actual position. The internal representation of a position is given by `Column-Row` (The row with index 0 in the board is the first row displayed). The representation of a move is given by `StartPosition-EndPosition`.

The conversion between I/O and move is achieved using the `parse_move` predicate and between move and positions is made using `parse_square`.

Output

The several messages throughout tJogadashe game are made relying on the `write` and `format` predicates.

Game settings are diplayed as such:

```
          _                       __  __
 _ | |___ ___ ___ _ _   |  \/  |___ _ _
| || / -_|_-</ _ \ ' \  | |\/| / _ \ '_|
 \__/\___/__/\___/_||_| |_|  |_\___/_|

 Bruno Mendes                up201906166@edu.fe.up.pt
 Jose Costa                  up201907216@edu.fe.up.pt

 ( Input the option number to select it )
 Choose the type of player for White:
       1. Human Player
       2. Sloppy Computer
       3. Greedy Computer
       4. Smart Computer
|: <your-input>.

 ( Input the option number to select it )
 Choose the type of player for Black:
       1. Human Player
       2. Sloppy Computer
       3. Greedy Computer
       4. Smart Computer
|: <your-input>.
 Input the desired board size (max.9, must be odd)
|: <your-input>.
```

The board is displayed in the following configuration:

```
  ---------------------
9 | b o o o o o b o b |
8 | o b o b o b o o o |
7 | o o o o o b o o o |
6 | o o b o o o o o o |
5 | o o o o b o o o o |
4 | o o o o o o o o o |
3 | o o o o o o o o o |
2 | w o o o o o w o w |
1 | w w o w w o o w o |
  ---------------------

Static board evaluation: -1.07
```

```
White to play
```

The board display predicate, `display_game(+GameState)`, is flexible printing boards as big as the boards inside the game state. The white pieces are represented in blue and the black pieces in red, when in game.

Following it is displayed a prompt that varies depending on the player nature (Human or Computer) Human:

```
Greedy Computer would play: c1-a2
( Input your move in algebraic notation as above )
|: <your-input>.
```

Computer:

```
Computer will play: d1-c3
(Press return)
```

The games finishes with the following display:

```
       Black has won the game! Congratulations!

           _                        __  __
 _ | |___ ___ ___ _ _   |  \/  |___ _ _
| || / -_|_-</ _ \ ' \  | |\/| / _ \ '_|
 \__/\___/__/\___/_||_| |_|  |_\___/_|

 Bruno Mendes                 up201906166@edu.fe.up.pt
 Jose Costa                   up201907216@edu.fe.up.pt


yes
```

The first `GameState` is generated by the `initial_state(+Size, -GameState)` predicate that, similarly to the `display_game` predicate, also generates a new state with a board as big as we might wish, even though we limited it to only print boards as big as 9x9 cells, due to performance issues when using bots in the game. The new state has the first player as white, as in chess.

Move execution

After correctly parsed by the `parse_move` predicate, the move is fed to the `move(+GameState, +Move, -NewGameState)` predicate that executes the move, relying on the `can_move` predicate to verify the validity of the move and in the `replace_nested` predicate to actually generate a new `Board` with the correct atoms in place. The new `GameState` is then assembled.

Valid moves

Making use of the same predicate as the `can_move` predicate uses to verify if the move of that piece is valid, we can generate a list of valid moves.

The `knight_move(+Move)` predicate generates all the possible moves or possible origins of a piece in a position by using backtrack. That is exactly what the `valid_moves(+GameState, -ListOfMoves)` predicate does, lsiting all the possible moves for a given `GameState`.

Game bots

Position scoring

`Jeson Mor` is a chess variant and we all know how complicated chess is as a game. It has been the same for more than 500 years and it's still competitive at the high level with the advent of computers.

For our bot (non random) to find a good move, it was necessary to implement a form of evaluating a position: no memory could hold the hundreds of millions of variations a game can have until its end. Chess engines like `Stockfish` evaluate a position by many criteria, retrieving a relative score:

- <0: black advantage (the lesser the better)
- Around 0: equality
- >0: white advantage (the greater the better)

We imitated this behaviour in our `board_evaluation` predicate. Let's check it:

```
board_evaluation([CP,CB,_], E):-
    material_count(w, CB, WM),
    material_count(b, CB, BM),
    WM > 0, BM > 0,
    knights_position_score(w, CB, WKS),
    knights_position_score(b, CB, BKS),
    attacking_knights([w,CB,_], WAK),
    attacking_knights([b,CB,_], BAK),
    ((CP = w, WAK > 0) -> PBAK is BAK - 1; PBAK is BAK),
    ((CP = b, BAK > 0) -> PWAK is WAK - 1; PWAK is WAK),
    E is WM + 0.25*(WKS/WM) + 0.25*PWAK - BM - 0.25*(BKS/BM) - 0.25*PBAK.
```

We evaluated a position by three criteria:

- Material difference: the most obvious and most valuable. A player with more pieces holds more chances to win the game.
- Tactical opportunity: the number of knights each player has attacking at least an opponent knight. For example, in a position where a white knight is attacked by two different black knights, black is likely to win the exchange and thus scores better in this criteria. Also, in a challenge of pieces, it's like the current player held one more knight, because it's the first to move.
- Strategic placement: knights are stronger close to the center since they can reach more squares and possibly get to the center square and leave, winning the game.

Algorithms

Having developed the heuristic, it was time to develop the algorithm to use it. We were only asked to implement a greedy computer, and in our testing, with the above heuristic, the play was decent, with the computer having the capacity to capture available opponent pieces, not blunder pieces straight away (due to the tactical opportunity factor) and improve the position of the knights when no piece challenges were in place. This was indeed sufficient to beat an average human player.

The greedy algorithm is actually a minimax with depth of 1: all possible moves are made, the resulting game states are evaluated and the move that maximizes the score in the perspective of calling player is chosen. This obviously lacks the ability to calculate deeply into a piece challenge, where several knights are defending a square, and several knights are attacking it.

To solve this limitation, we implemented the complete minimax algorithm. Let's analyse it:

```
minimax_aux(1, [CP,CB,_], M-E):-
    valid_moves([CP,CB,_], ML),
    maplist(move([CP,CB,_]), ML, NGS),
    maplist(value(CP), NGS, EL),
    max_element(EL, I, _), % depth 1 is always the maximizer
    nth0(I, ML, M),
    nth0(I, EL, E).
minimax_aux(D, [CP,CB,_], M-E):-
    D > 1,
    ND is D - 1,
    valid_moves([CP,CB,_], ML_),
    maplist(move([CP,CB,_]), ML_, NGS),
    maplist(minimax_aux(ND), NGS, MSL),
    maplist(filter_score_minimax, MSL, SL),
    % odd depths lead to us being the maximizer at the base case
    % even depths lead to the opponent
    (D mod 2 =:= 0 -> min_element(SL, I, E); max_element(SL, I, E)),
    nth0(I, ML_, M).
```

Depth 1 is the leaf of all branches. Getting to this base case, the position is evaluated using the heuristic and the score is returned to the parents. We decided this was going to be a maximizer: it's up to the parents to know whether they are maximizers or minimizers according to the parity of the depth.

The space of all game states is large. With 9 knights each side, each capable of 8 moves (let's say they are far off the board borders), we already have 72 possible next positions. If no captures are made, the number of possible positions for the next n moves is 72^n. We tried to prune the calculations by resetting very imbalanced positions to a greedy evaluation and skipping pointless calculations

of won games:

```
minimax_aux(D, [CP,CB,OB], _-E):-
    game_over([CP,CB,OB],_), !, % cut branch if game is over
    value(CP, [CP,CB,OB], E_),
    (D mod 2 =:= 0 -> E is -E_; E is E_). % simulate this being the base case
minimax_aux(D, [CP,CB,OB], M-E):-
    D > 1, D mod 2 =\= 0, % only jump if base case is same player
    value(CP, [CP,CB,OB], E_),
    E_ > 2, !, % cut branch if static board evaluation is obvious
    minimax_aux(1, [CP,CB,OB], M-E).
```

This has proven to have little effect on game openings, where the position is equal and the game is far from won for either side. A possible improvement is the use of alpha-beta pruning, an already explored topic in the chess programming community we would have liked to explore if we had find the time to do so.

Nevertheless, the reader can play with our raw minimax implementation, with low depths (the default from the menu is 3) and small boards, or not, if they can grab a cup of coffee. A computer vs computer game is also a fun experience, with both bots eventually reaching a dead end and not engaging in challenges they know they cannot calculate (this would be a draw in chess, but no such rule exists for `Jeson Mor`).

As a last note, the reader also gets a quick hint from the greedy computer when playing in human mode, in case they feel lost.

Execution flow

The program, after being laucnhed with the `play` predicate, uses the `retractall` predicate for `player/2` and `state/1` to posssibly clear data from previous games.

Then summons the menu that prompts the user for the game settings (Nature of each player, Human or Computer) and board size. To keep this choices and save the initial game state generated, makes use of `assertz` predicates.

From here sets up a fail based loop with `repeat`. The loop that will be responsible for checking for the game ending (and if so, for displaying the congratulations message and clearing the knowledge base of asserted predicates, exiting the loop and allowing for progam termination), and performing operations to control the game, generating a new `GameState`, retracting (`retract`) the old state from the knowledge base and saving the new state (`assert`), before failing (`fail`) and using backtrack to reenter the loop.

The operations of the game are displaying the board to the user and retrieving moves (in case of a Human player) or confirmation (in case of a Computer player).

Conclusion

For imperative minded programmers, Prolog proves to be an healthy challenge, forcing us to focus on the problem instead of in the steps to solve it. This isn't always easy, and we found ourselves battling many times for thinking imperatively, gasping for cycles, when declarative programming offered us a compact and logical solution. More often than not, long hours translated in few but powerful lines of code.

Known Issues

- Color output in Windows - We used ANSI espace codes for outputting colored pieces, which may not be available in proprietary or old Windows consoles.