Neutreeko

Final Report

Mestrado Integrado em Engenharia Informática e Computação

Programação em Lógica

Grupo Neutreeko: 1
Bernardo Manuel Costa Barbosa - up201503477
João Pedro Teixeira Pereira de Sá - up201506252

Faculdade de Engenharia da Universidade do Porto
Rua Roberto Frias, sn, 4200-465 Porto, Portugal

November 18, 2018

Contents

1 Introduction

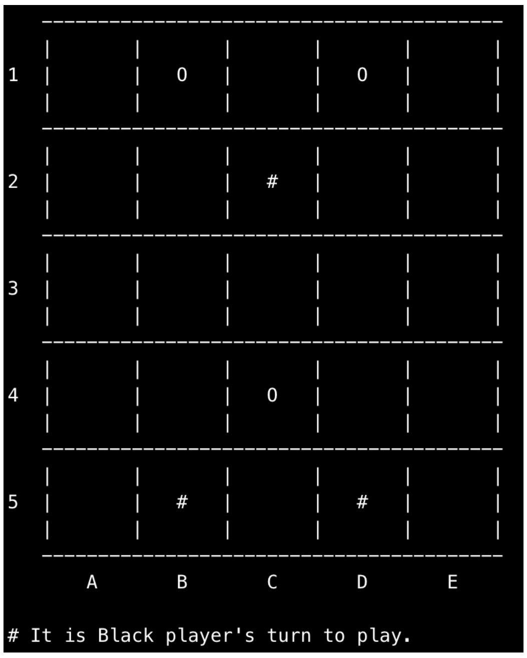
The goal of this work is to implement a game called Neutreeko in Prolog using the SICStus interpreter, in the context of the discipline Program in Logic of the Integrated Master in Computer Engineering and Computing.

2 Neutreeko

Neutreeko is a simple game played on a board with 5×5 squares.

It's a two-player abstract board game invented by Jan Kristian Haugland in 2001 and is a portmanteau of Neutron and Teeko, two games on which it is based.

Each player starts with three pieces each, as shown in the figure.



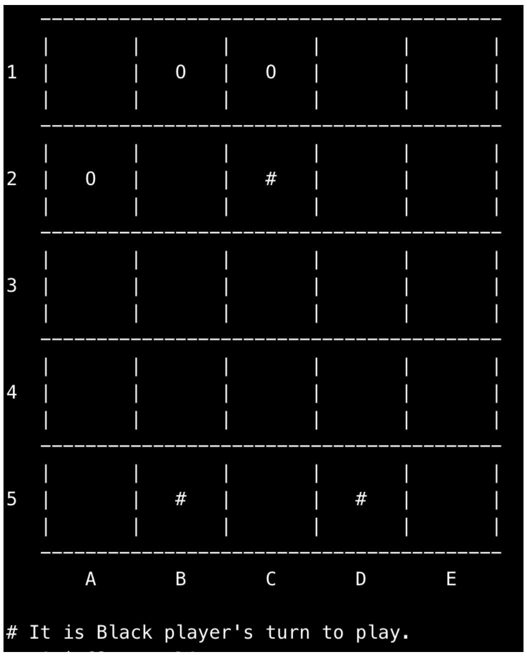
```
    -----------------------------------------
    |       |       |       |       |       |
1   |       |   O   |       |   O   |       |
    |       |       |       |       |       |
    -----------------------------------------
    |       |       |       |       |       |
2   |       |       |   #   |       |       |
    |       |       |       |       |       |
    -----------------------------------------
    |       |       |       |       |       |
3   |       |       |       |       |       |
    |       |       |       |       |       |
    -----------------------------------------
    |       |       |       |       |       |
4   |       |       |   O   |       |       |
    |       |       |       |       |       |
    -----------------------------------------
    |       |       |       |       |       |
5   |       |   #   |       |   #   |       |
    |       |       |       |       |       |
    -----------------------------------------
        A       B       C       D       E

# It is Black player's turn to play.
```

Figure 1: Initial State of Neutreeko.

Rules:

- Black always moves first.
- A piece slides orthogonally or diagonally until stopped by an occupied square or the border of the board.

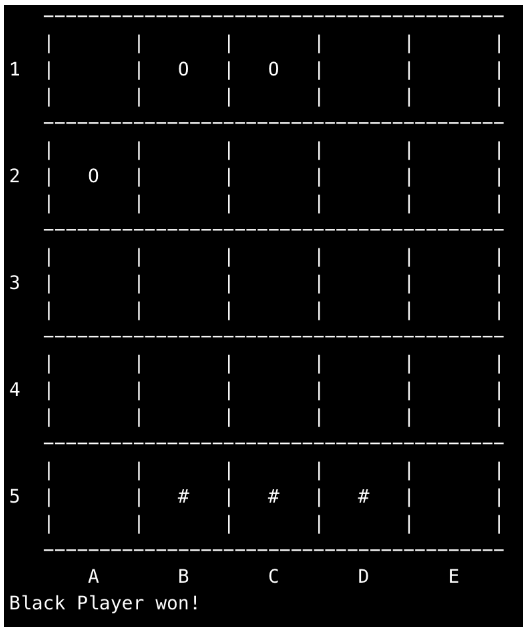


```
    -----------------------------------------
    |       |       |       |       |       |
1   |       |   O   |   O   |       |       |
    |       |       |       |       |       |
    -----------------------------------------
    |       |       |       |       |       |
2   |   O   |       |   #   |       |       |
    |       |       |       |       |       |
    -----------------------------------------
    |       |       |       |       |       |
3   |       |       |       |       |       |
    |       |       |       |       |       |
    -----------------------------------------
    |       |       |       |       |       |
4   |       |       |       |       |       |
    |       |       |       |       |       |
    -----------------------------------------
    |       |       |       |       |       |
5   |       |   #   |       |   #   |       |
    |       |       |       |       |       |
    -----------------------------------------
        A       B       C       D       E

# It is Black player's turn to play.
```

Figure 2: Example of Intermediate State of Neutreeko.

- The goal of Neutreeko is to place three of your own checkers in a row, orthogonally or diagonally. The row must be connected.
- The game is declared a draw if the same position occurs three times.



```
    -----------------------------------------
    |       |       |       |       |       |
1   |       |   O   |   O   |       |       |
    |       |       |       |       |       |
    -----------------------------------------
    |       |       |       |       |       |
2   |   O   |       |       |       |       |
    |       |       |       |       |       |
    -----------------------------------------
    |       |       |       |       |       |
3   |       |       |       |       |       |
    |       |       |       |       |       |
    -----------------------------------------
    |       |       |       |       |       |
4   |       |       |       |       |       |
    |       |       |       |       |       |
    -----------------------------------------
    |       |       |       |       |       |
5   |       |   #   |   #   |   #   |       |
    |       |       |       |       |       |
    -----------------------------------------
        A       B       C       D       E
Black Player won!
```

Figure 3: Example of Final State of Neutreeko.

3 Game Logic

3.1 Game State Representation

We have implemented the following predicate:

```
initial_board([
  [emptyCell,whitePiece,emptyCell,whitePiece,emptyCell],
  [emptyCell,emptyCell,blackPiece,emptyCell,emptyCell],
  [emptyCell,emptyCell,emptyCell,emptyCell,emptyCell],
  [emptyCell,emptyCell,whitePiece,emptyCell,emptyCell],
  [emptyCell,blackPiece,emptyCell,blackPiece,emptyCell]
  ]).
```

This predicate contains a list of lists, in which each sublist represents a line of the board, and each element of that same sublist represents a position on the line (column).

All predicates constructed for manipulation of the list have as input the letter and number coordinates, starting with A and 1.

To represent the parts in the list, each atom of the sublists can have the following value:

- emptyCell - Empty Cell;
- blackCell - Black Piece;
- whiteCell - White Piece;

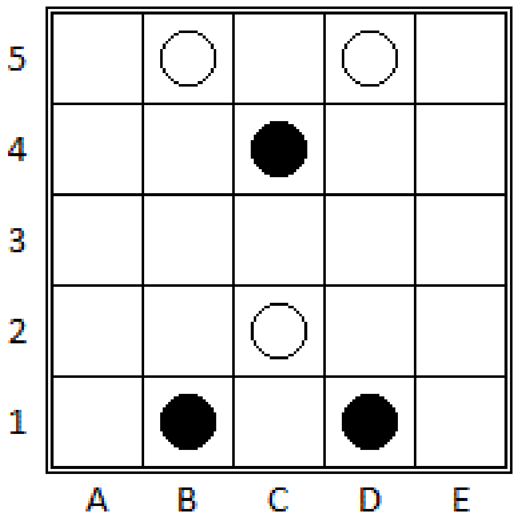


Figure 4: Example of Neutreeko Game.

3.2 Board Visualization

For the visualization of the board, we implemented two identical functions, one to print the board during the game (which receives the player as an argument) and the other to print the final state of the game.

The display_game / 1 predicate is used to represent the board in the console during the game.

```
% display_game(+Board)
display_game(Board):-
  display_board(Board, 0),
  write_line, nl,
  write_letters,nl, !.
```

The display_game / 2 predicate is used to represent the board in the final state of the game.

```
% display_game(+Board, +Player)
display_game(Board, Player):-
  display_board(Board, 0),
  write_line, nl,
  write_letters,nl,
  print_turn_info(Player), !.
```

Both predicates begins by writing the board by going to display_board /2. Then it goes to the write_line predicate and finally to write_letters to get the representation of the coordinates.

The remaining recursive predicates are represented here:

```
% display_game(+Board, +Y)
display_board([],_).
display_board([Line|Tail], Y):-
  write_line, nl,
  write_spaces, nl,
  Y1 is Y+1,
  display_line(Line, Y1), nl,
  write_spaces, nl,
  display_board(Tail, Y1).

% display_line(+Board, +Line)
display_line(Line, Y):-
  write(Y), write(' |'),
  display_line_aux(Line).

% display_line_aux(+List)
display_line_aux([]).
display_line_aux([Cell|Tail]):-
  get_cell_symbol(Cell,Symbol),
  write('  '), write(Symbol), write(' |'),
  display_line_aux(Tail).
```

Since we are sending each row of the array to this predicate, we simply check the type of part and write the representation of the part to the screen, according to the following conversion:

- emptyCell - ' ';
- blackCell - #;
- whiteCell - O;

3.3 List of Valid Moves

To get all the possible valid moves of a player, we implement the predicate valid_moves (+ Board, + Player, -ListOfMoves), which basically tests all the valid moves for each player piece and returns a list of moves that is later used for the player to choose the move he wants to make.

```
% Function that gets all player's valid movements.
% valid_moves(+Board, +Player, -ListOfMoves)
valid_moves(Board, Player, ListOfMoves):-
  get_player_piece(Player, Piece),
  findall(m(Yi, Xi, Y, X),
    (
       getMatrixElemAt(Yi, Xi, Board, Piece),
       between(0, 4, Y), between(0, 4, X),
       validate_move(m(Yi, Xi, Y, X), Board)
    ),
    ListOfMoves).
```

To validate a movement, we have verified whether this movement (horizontal, vertical or diagonal) complies with the rules of the game. In this piece of code, it is possible to observe the main function together with the verification for a horizontal movement, being identical to the other two types of movement, varying only the direction:

```
% Function that validates a move.
% validate_move(+Move, +Board)
validate_move(m(Yi, Xi, Yf, Xf), Board):-
  validate_X_move(m(Yi, Xi, Yf, Xf), Board);
  validate_Y_move(m(Yi, Xi, Yf, Xf), Board);
  validate_XY_move(m(Yi, Xi, Yf, Xf), Board).

% Functions that validates a horizontal move.
% validate_X_move(+Move, +Board)
validate_X_move(m(Yi, Xi, Yf, Xf), Board):-
  DiffX is Xf - Xi,
  DiffY is Yf - Yi,
  DiffY == 0, DiffX \= 0,
  (DiffX > 0 -> Direction is 1 ; Direction is -1),
  validate_X_move_aux(m(Yi, Xi, Yf, Xf), Board, Direction, 0).

% validate_X_move_aux(+Move, +Board, +Direction, +CurrIndex)
validate_X_move_aux(m(Yi, Xi, Yf, Xf), Board, Direction, CurrIndex):-
  NewIndex is CurrIndex+Direction,
  NextX is Xi+NewIndex,
  getMatrixElemAt(Yi, NextX, Board, NextElem),
  (
    NextElem == 'emptyCell' ->
        (NextX == Xf ->
          (AfterX is NextX+Direction,
           getMatrixElemAt(Yi, AfterX , Board, NextElem) ->
            (NextElem == 'emptyCell' -> false ; true) ; true);
            validate_X_move_aux(m(Yi, Xi, Yf, Xf), Board, Direction, NewIndex)
        ) ; false
  ).
```

3.4 Board Movement

To execute a move on the board, the predicate move(+ Move, + Board, -NewBoard) is preceded by the validate_move/2, already discussed above.

```
% Function that makes a Player move on the Board.
% move(+Move, +Board, -ResultantBoard)
move(m(Yi, Xi, Yf, Xf), Board, ResultantBoard):-
  getMatrixElemAt(Yi, Xi, Board, SrcElem),
  setMatrixElemAtWith(Yi, Xi, emptyCell, Board, TempBoard),
  setMatrixElemAtWith(Yf, Xf, SrcElem, TempBoard, ResultantBoard).
```

3.5 Game Over

To determine if the game is over, a check is made on the board before any movement is made to check for a tie (third replay of the pieces) or win (3 consecutive pieces horizontally, vertically or diagonally). To do this, we implemented the predicate game_over (+ Board, -Winner) together with the functions that help to check the direction of the 3 pieces and the functions that verify the tie.

```
% Functions that check if the game is tied.
game_tie:-
  game_board_history(BoardHistory),
  check_tie(BoardHistory, BoardHistory).

check_tie([Head|_], BoardHistory):-
  count(BoardHistory, Head, NOfOccur),
  NOfOccur >= 3.

check_tie([Head|Tail], BoardHistory):-
  count(BoardHistory, Head, NOfOccur),
  NOfOccur < 3,
  check_tie(Tail, BoardHistory).

game_winner(Winner):-
  game_board(Board),
  game_over(Board, Winner).

% Function that checks if a board has a winner.
% game_over(+Move, -Winner)
game_over(Board, Winner) :-(
  checkVertical(Board, Piece) ;
  checkHorizontal(Board, Piece) ;
  checkDiagonal(Board, Piece)), (
  Piece == 'blackPiece' -> Winner = 'blackPlayer' ;
  (Piece == 'whitePiece' -> Winner = 'whitePlayer' ; false)).

% Function that checks if there are 3 consecutive pieces horizontally.
% checkHorizontal(+Move, +Piece)
checkHorizontal(Board, Piece) :-
  getMatrixElemAt(X, Y, Board, Piece),
  piece(Piece),
  Y1 is Y+1, Y2 is Y+2,
  getMatrixElemAt(X, Y1, Board, Elem2),
  getMatrixElemAt(X, Y2, Board, Elem3),
  Piece == Elem2, Piece == Elem3.

% Function that verifies that there are 3 consecutive pieces vertically.
% checkVertical(+Move, +Piece)
checkVertical(Board, Piece) :-
  getMatrixElemAt(X, Y, Board, Piece),
  piece(Piece),
  X1 is X+1, X2 is X+2,
  getMatrixElemAt(X1, Y, Board, Elem2),
  getMatrixElemAt(X2, Y, Board, Elem3),
  Piece == Elem2, Piece == Elem3.

% Function that checks if there are 3 consecutive pieces diagonally.
% checkDiagonal(+Move, +Piece)
checkDiagonal(Board, Piece) :- (
  getMatrixElemAt(X, Y, Board, Piece),
  piece(Piece),
  X1 is X+1, X2 is X+2, Y1 is Y+1, Y2 is Y+2,
  getMatrixElemAt(X1, Y1, Board, Elem2),
  getMatrixElemAt(X2, Y2, Board, Elem3),
  Piece == Elem2, Piece == Elem3
  );
  (getMatrixElemAt(X, Y, Board, Piece),
  piece(Piece),
  X1 is X+1, X2 is X+2, Y1 is Y-1, Y2 is Y-2,
  getMatrixElemAt(X1, Y1, Board, Elem2),
  getMatrixElemAt(X2, Y2, Board, Elem3),
  Piece == Elem2, Piece == Elem3
  ).
```

3.6 Board Rating

Since originally we planned to implement alpha-beta reliant AI, we developed a complex method of evaluating a board. The predicate value(+Board, +Player, -Value) unifies a value that can either be positive (good for the maximizer player) or negative (good for the minimizer player).

What we are trying to achieve here is:

- evaluate the pieces position.
- evaluate if any player has two pieces in a row, with the possibility of having three.
- generate a small random variation.
- evaluate any end-game scenario.

```
% Function that calculates board value.
% value(+Board, -Value)
value(Board, Value):-
  evaluate_board_pieces(Board, PiecesValue, 0),
  evaluate_board_state(Board, StateValue, 0),
  generate_random_component(RandomValue),
  evaluate_game_over(Board, GameOverValue),
  Value is PiecesValue + StateValue + RandomValue + GameOverValue.

% evaluate_board_pieces(+Board, -Value, +RowIterator)
evaluate_board_pieces(_, 0, RowIterator):-
  RowIterator>4.

evaluate_board_pieces(Board, Value, RowIterator):-
  nth0(RowIterator, Board, Line), !,
  evaluate_line_pieces(Line, LineValue, RowIterator, 0),
  RowIteratorNext is RowIterator+1,
  evaluate_board_pieces(Board, RemainingValue, RowIteratorNext),
  Value is LineValue+RemainingValue.

% evaluate_line_pieces(+Board, -LineValue, +RowIterator, +ColIterator)
evaluate_line_pieces(_, 0, _, ColIterator):-
  ColIterator>4.

evaluate_line_pieces(Line, LineValue, RowIterator, ColIterator):-
```

```
  nth0(ColIterator, Line, Piece), !,
  piece_value(Piece, PieceValue),
  board_weight(RowIterator, ColIterator, Weight),
  ColIteratorNext is ColIterator+1,
  evaluate_line_pieces(Line, RemainingValue, RowIterator, ColIteratorNext),
  LineValue is PieceValue*Weight + RemainingValue.

% evaluate_board_state(+Board, -Value, +RowIterator)
evaluate_board_state(_, 0, RowIterator):-
  RowIterator>4.

evaluate_board_state(Board, Value, RowIterator):-
  !,
  evaluate_line_state(Board, LineValue, RowIterator, 0),
  RowIteratorNext is RowIterator+1,
  evaluate_board_state(Board, RemainingValue, RowIteratorNext),
  Value is LineValue+RemainingValue.

% evaluate_line_pieces(+Board, -LineValue, +RowIterator, +ColIterator)
evaluate_line_state(_, 0, _, ColIterator):-
  ColIterator>4.

evaluate_line_state(Board, LineValue, RowIterator, ColIterator):-
  !,
  RowAbove is RowIterator+1, RowBelow is RowIterator-1, RowAboveAbove is RowIterator+2, RowBelowBelow
      is RowIterator-2,
  ColAhead is ColIterator+1, ColBehind is ColIterator-1, ColAheadAhead is ColIterator+2,
  %Check above
  ((getMatrixElemAt(RowIterator, ColIterator, Board, Piece1),
  getMatrixElemAt(RowAbove, ColIterator, Board, Piece2),
  (getMatrixElemAt(RowAboveAbove, ColIterator, Board, Piece3), Piece3 == emptyCell;
  getMatrixElemAt(RowBelowBelow, ColIterator, Board, Piece4), Piece4 == emptyCell),
  Piece1 \= emptyCell,
  Piece1 == Piece2)->
    piece_value(Piece1, PieceValue1), TempValue1 is PieceValue1 * 500;
    TempValue1 is 0
  ),
  %Check above right
  ((getMatrixElemAt(RowIterator, ColIterator, Board, Piece5),
  getMatrixElemAt(RowAbove, ColAhead, Board, Piece6),
  (getMatrixElemAt(RowAboveAbove, ColAheadAhead, Board, Piece7), Piece7 == emptyCell;
  getMatrixElemAt(RowBelow, ColBehind, Board, Piece8), Piece8 == emptyCell),
  Piece5 \= emptyCell,
  Piece5 == Piece6)->
    piece_value(Piece5, PieceValue2), TempValue2 is PieceValue2 * 500;
    TempValue2 is 0
  ),
  %Check right
  ((getMatrixElemAt(RowIterator, ColIterator, Board, Piece9),
  getMatrixElemAt(RowIterator, ColAhead, Board, Piece10),
  (getMatrixElemAt(RowIterator, ColAheadAhead, Board, Piece11), Piece11 == emptyCell;
  getMatrixElemAt(RowIterator, ColBehind, Board, Piece12), Piece12 == emptyCell),
  Piece9 \= emptyCell,
  Piece9 == Piece10)->
    piece_value(Piece9, PieceValue3), TempValue3 is PieceValue3 * 500;
    TempValue3 is 0
  ),
  %Check below right
  ((getMatrixElemAt(RowIterator, ColIterator, Board, Piece13),
  getMatrixElemAt(RowBelow, ColAhead, Board, Piece14),
  (getMatrixElemAt(RowBelowBelow, ColAheadAhead, Board, Piece15), Piece15 == emptyCell;
  getMatrixElemAt(RowAbove, ColBehind, Board, Piece16), Piece16 == emptyCell),
  Piece13 \= emptyCell,
  Piece13 == Piece14)->
    piece_value(Piece13, PieceValue4), TempValue4 is PieceValue4 * 500;
    TempValue4 is 0
  ),
  evaluate_line_state(Board, RemainingValue, RowIterator, ColAhead),
  LineValue is TempValue1 + TempValue2 + TempValue3 + TempValue4 + RemainingValue.

% generate_random_component(-RandomValue)
generate_random_component(RandomValue):-
  random(-10, 11, RandomValue).

% evaluate_game_over(+Board, -GameOverValue)
evaluate_game_over(Board, GameOverValue):-
  (board_winner(Board, Winner) ->(
    Winner == 'blackPlayer' -> piece_value(blackPiece, PieceValue), GameOverValue is PieceValue * 100000;
    Winner == 'whitePlayer' -> piece_value(whitePiece, PieceValue), GameOverValue is PieceValue * 100000
    );
    GameOverValue is 0
  ).
```

3.7 Computer's Move

For the computer, we assign two types of difficulty: greedy and dumb. The dumb mode consists of randomly choosing one of the possible valid moves, while the greedy mode consists of walking through every possible move the bot can choose, assigning them a value accordingly to the resulting board, and choosing the lowest (minimizer) or highest (maximizer). The predicate choose_move (+Board, +Player, +Level, -Move) returns the move as the difficulty is set.

```
% Function that chooses the next movement of the computer depending on its difficulty.
% choose_move(+Board, +Player, +Level, -Move)
choose_move(Board, Player, Level, Move):-
  valid_moves(Board, Player, ListOfMoves),
  (
    Level == random -> dumb_bot(Board, ListOfMoves, Move);
    greedy_bot(Board, ListOfMoves, Player, Move)
  ).

% Dumb Computer logic.
% dumb_bot(+Board, +ListOfMoves, -Move)
dumb_bot(Board, ListOfMoves, Move):-
  length(ListOfMoves, Length),
  random(0, Length, MoveIndex),
  nth0(MoveIndex, ListOfMoves, Move).

% Function that evaluates all the board values for each possible movement.
% evaluate_list_of_moves(+Board, +ListOfMoves, -ListValueOfMoves)
evaluate_list_of_moves(_, [], []).
evaluate_list_of_moves(Board, [Head|Tail], ListValueOfMoves):-
  evaluate_list_of_moves(Board, Tail, TailListValueOfMoves),
  move(Head, Board, ResultantBoard),
  value(ResultantBoard, BoardValue),
  append(TailListValueOfMoves, [BoardValue], ListValueOfMoves).

% choose_min_move(+ListValueOfMoves, -MinimizerMoveIndex)
choose_min_move(ListValueOfMoves, MinimizerMoveIndex):-
  min_member(MinValue, ListValueOfMoves),
  nth0(MaximizerMoveIndex, ListValueOfMoves, MinValue).

% choose_max_move(+ListValueOfMoves, -MaximizerMoveIndex)
choose_max_move(ListValueOfMoves, MaximizerMoveIndex):-
  max_member(MaxValue, ListValueOfMoves),
  nth0(MaximizerMoveIndex, ListValueOfMoves, MaxValue).
```

```
% Greedy Computer logic.
% greedy_bot(+Board, +ListOfMoves, +Player, -Move)
greedy_bot(Board, ListOfMoves, Player, Move):-
  evaluate_list_of_moves(Board, ListOfMoves, ListValueOfMoves),
  ((
    maximizing(Player),
    choose_max_move(ListValueOfMoves, MoveIndex)
  );
  (
    minimizing(Player),
    choose_min_move(ListValueOfMoves, MoveIndex)
  )),
  nth0(MoveIndex, ListOfMoves, Move).
```

4 Conclusions

The group is satisfied with the work done, we have succeeded in accomplishing everything we have committed to and delivered a competitive and interactive game.

Overall, both elements worked hard not only to develop the game but also to solve problems that arose during the implementation of various functionalities.

For us it was a challenge to develop a game in a new language like Prolog, it forced us to adopt a generative and recursive approach, but we thought that in the end we managed to live up to expectations.

5 References

http://www.neutreeko.net/neutreeko.htm
http://www.iggamecenter.com/info/en/neutreeko.html
https://sicstus.sics.se/documentation.html