# Apart - G11_02

## Group Members

| Nome | Up | Contribuição |
|---|---|---|
| Inês Alexandre Queirós Matos Macedo de Oliveira | 202103343 | 50% |
| João Pedro Cruz Moreira de Oliveira | 202108737 | 50% |

## Installation and Execution

To be able to open the game, you need to install SICStus Prolog (latest version) and download the source code. Once you have opened the program, select "File" (top left), "browse" and finally select the "main.pl" file.

Finally, execute the "play." command.

## Description of the game

*Apart* is a strategy game for two players that takes place on a square board, usually of odd size, such as 8x8 or larger (up to 25x25). The board starts with 12 pieces on each side (6 in each row). Player 1's pieces are identified by the "W" symbol, while player 2's pieces are represented by the "B" symbol. The aim of the game is to be the first player to position all their pieces vertically, horizontally and diagonally free from adjacent pieces of the same color. If both players reach this state simultaneously, the player who made the last move loses. The game has many interesting move restrictions: - Moving a piece to an adjacent square vertically, horizontally or diagonally is called a single step, and moving two or more is called a jump. - Start with the player with white color and play alternately. - Only the first move of the first player cannot be a continuous jump. - All pieces can be moved in the direction along the line to which they belong, for a distance equal to the length of that line. - Pieces in between can be jumped over, and if there is an enemy piece at the landing site, it is captured and removed from the game. - Pieces cannot be moved to the location of a friendly piece. - If the jumped piece can jump again from the position it has moved to, it can continue to jump in the same turn. - There is no limit to the number of jumps, but the same square may not be landed twice in the same turn. - A single step cannot be included in a sequence of a continuous jump

The game involves strategic planning and tactics to keep good movement options available while trying to win the game. It's a game that challenges players' ability to anticipate and strategize, making each match unique and exciting.

## Game Logic

### Internal Game State Representation

The 'GAMESTATE' constitutes an argument presented in nearly every primary predicate within the game. It includes the following elements:

- **BOARD:** It is a matrix with a size determined by the user. Initially, it contains player 2's pieces represented by "B" at the top of the game, divided into two rows, the number of pieces being SIZE-2 per row. The same happens at the bottom of the game, for player 1, with his pieces represented by "W".
- **SIZE:** The board size that the user chooses before starting the game.
- **PLAYER:** The player that will make the next move.
- **GAMEMODE:** The gamemode that the user chooses before starting the game.
- **BOTLEVEL:** The bot level that the user chooses before starting the game. If the user chooses gamemode "h/h", bot level has default value of 1.

To build the 'GAMESTATE' after the user chooses its elements, we use a predicate **initial_state(+SIZE, -GAMESTATE)** that receives the board size as an argument and returns an initial game state.

For the following examples, we will use a board size of 8 and gamemode 'h/h'.

### Initial Game State

```
GAMESTATE(
    [
        [0, 1, 1, 1, 1, 1, 1, 0],
        [0, 1, 1, 1, 1, 1, 1, 0],
        [0, 0, 0, 0, 0, 0, 0, 0],
        [0, 0, 0, 0, 0, 0, 0, 0],
        [0, 0, 0, 0, 0, 0, 0, 0],
        [0, 0, 0, 0, 0, 0, 0, 0],
        [0, 2, 2, 2, 2, 2, 2, 0],
        [0, 2, 2, 2, 2, 2, 2, 0]
    ], % BOARD
    8, % SIZE
    'W', % PLAYER
    h/h, % GAMEMODE
    1, % BOTLEVEL
)
```

*Image 1: Initial Game State*

### Intermediate Game State

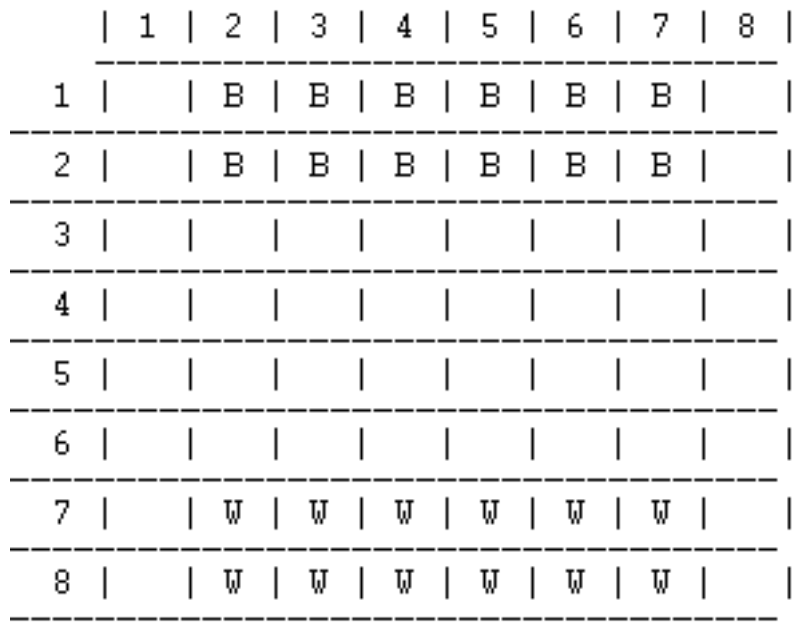


Figure 1: image.png

```
GAMESTATE(
    [
        [1, 0, 0, 0, 0, 1, 1, 0],
        [1, 0, 1, 0, 0, 0, 0, 0],
        [0, 0, 0, 0, 0, 0, 0, 0],
        [0, 1, 0, 1, 0, 1, 0, 0],
        [0, 2, 0, 2, 0, 0, 0, 0],
        [2, 0, 0, 0, 2, 0, 0, 2],
        [0, 0, 0, 0, 0, 0, 0, 0],
        [0, 0, 2, 2, 0, 0, 0, 0]
    ], % BOARD
    8, % SIZE
    'B', % PLAYER
    h/h, % GAMEMODE
    1, % BOTLEVEL
)
```

*Image 2: Intermediate Game State*

**Final Game State**

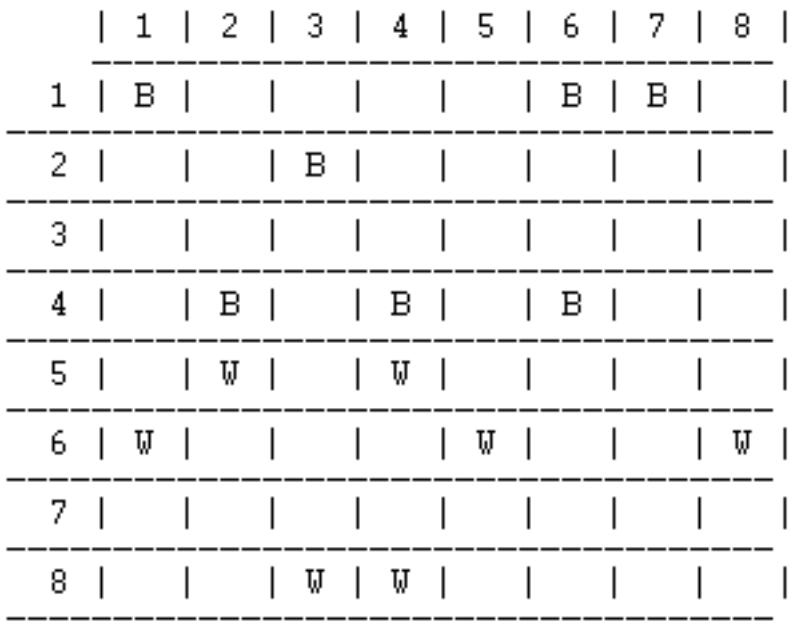


Figure 2: image.png

```
GAMESTATE(
    [
        [1, 0, 0, 0, 0, 1, 0, 0],
        [0, 0, 1, 0, 0, 0, 0, 1],
        [0, 0, 0, 0, 0, 0, 0, 0],
        [0, 1, 0, 1, 0, 1, 0, 0],
        [0, 2, 0, 2, 0, 0, 0, 0],
        [2, 0, 0, 0, 2, 0, 0, 2],
        [0, 0, 0, 0, 0, 0, 0, 0],
        [2, 0, 0, 2, 0, 0, 0, 0]
    ], % BOARD
    8, % SIZE
    'B', % PLAYER
    h/h, % GAMEMODE
    1, % BOTLEVEL
)
```

*Image 3: Final Game State*

### Game State Visualization

Before starting the game, the user(s) must prepare the match. To do this, several menus are presented, in which all the answers are validated(*), checking that

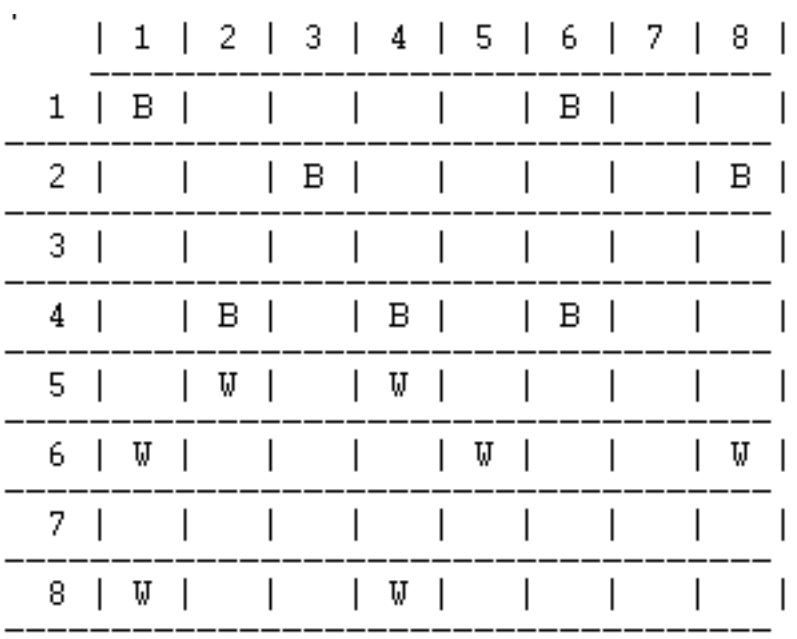


Figure 3: image.png

they are within the range of options and in all the menus there is the option to return to the initial menu.

(*) Input validation code:

```
get_menuinput(LOWERBOUND, UPPERBOUND, INPUT) :-
    repeat,
    format('Please choose an option between ~w and ~w:', [LOWERBOUND, UPPERBOUND]),
    (catch(read(OPTION), _, fail), between(LOWERBOUND, UPPERBOUND, OPTION) ->
        INPUT = OPTION
    ;
        format('Invalid option. Option is not in the range ~w to ~w.\n', [LOWERBOUND, UPPERB
        fail
    ), !.
```

```
// File: io.pl
```

In the Game menu you can choose between 4 game modes, where you can play with 2 humans, a human and the computer or the computer against the computer. Selecting any of the options brings up the board size menu.

*Image 4: Game Mode Menu*

After the user chooses an option, the gamemode is dynamically introduced in

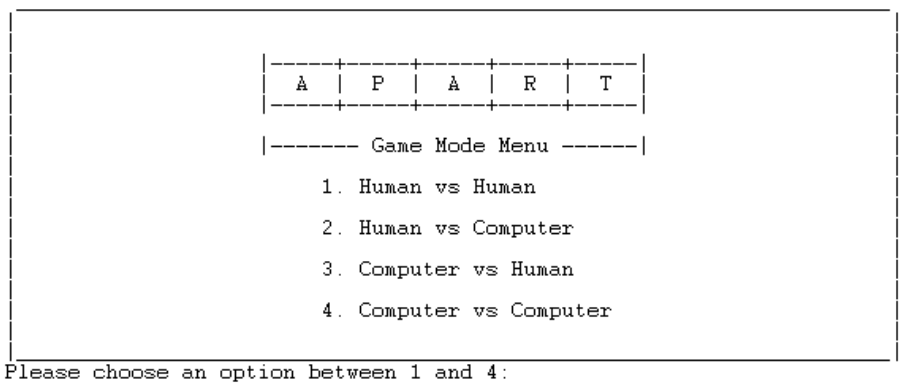


Figure 4: image.png

the fact base as follows:

```
set_gamemode(1) :-
    retract(gamemode(_)),
    assert(gamemode(h/h)).
set_gamemode(2) :-
    retract(gamemode(_)),
    assert(gamemode(h/c)).
set_gamemode(3) :-
    retract(gamemode(_)),
    assert(gamemode(c/h)).
set_gamemode(4) :-
    retract(gamemode(_)),
    assert(gamemode(c/c)).
```

```
// File: io.pl
```

Selecting any of the options brings up the board size menu. Here, you can choose any board size between 8 and 25, and the larger the board, the harder it will be to finish the game.

*Image 5: Board Size Menu*

After the user chooses an option, the board size is dynamically introduced in the fact base as follows:

```
set_boardsize(0, _) :-
    main_menu.
set_boardsize(1, _) :-
    true.
set_boardsize(2, INPUT) :-
```

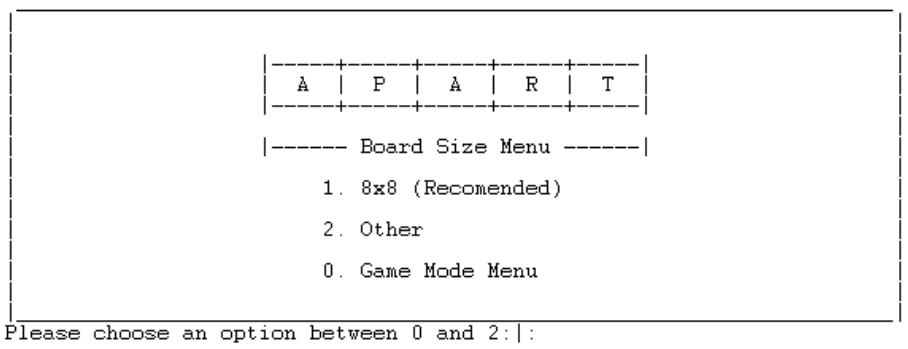


Figure 5: image.png

```
    retract(boardsize(_)),
    assert(boardsize(INPUT)).
```

```
// File: io.pl
```

Once you've determined the size of the board, you need to know the order in which you want to play. There are two options: play with the white pieces (W) and go first, or play with the black pieces (B) and go second.

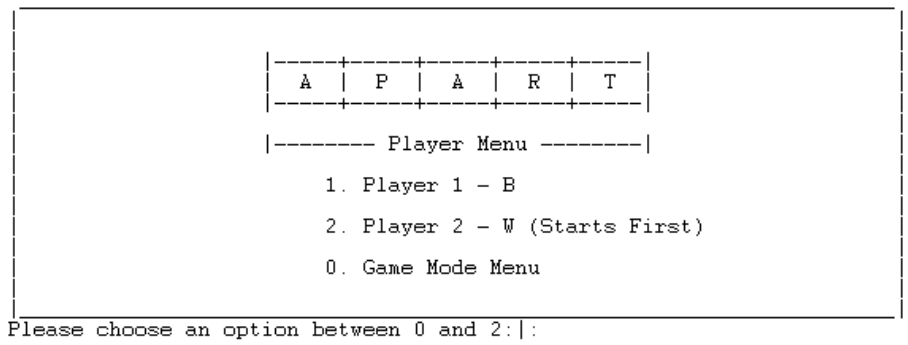


Figure 6: image.png

*Image 6: Player Menu*

After the user chooses an option, we use the following predicates:

```
set_playerside(0) :-
    main_menu.
set_playerside(1) :-
    true.
set_playerside(2) :-
```

```
    true.
```

```
// File: io.pl
```

Finally, if the answer in the Game mode menu was not 1 'Human vs Human', you will be presented with the Bot Menu, where you can choose the difficulty of the computer's moves. Thus, level 1 must return a valid random move. Level 2 should return the best move at the moment (using a greedy algorithm), taking into account the evaluation of the game state.

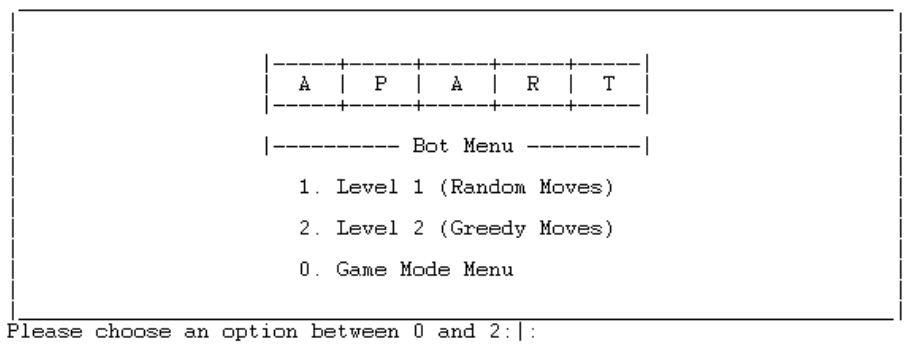


Figure 7: image.png

*Image 7: Bot Menu*

After the user chooses an option, the bot level is dynamically introduced in the fact base as follows:

```
set_bot_level(0) :-
    main_menu.
set_bot_level(1) :-
    true.
set_bot_level(2) :-
    retract(bot_level(_)),
    assert(bot_level(2)).
```

```
// File: io.pl
```

Once all the options have been chosen, the **initial_state(+SIZE, -GAMESTATE)** predicate is used to create the initial game state.

```
initial_state(SIZE, GAMESTATE) :-
    build_board(SIZE, BOARD),
    gamemode(GAMEMODE),
    bot_level(BOTLEVEL),
```

```
    GAMESTATE = [BOARD, SIZE, 'W', GAMEMODE, BOTLEVEL].
```

```
// File: game.pl
```

To prepare the game and start the game cycle, we used the **prepare_game/0** predicate.

```
prepare_game :-
    cs,
    boardsize(SIZE),
    initial_state(SIZE, GAMESTATE),
    change_random_seed,
    display_game(GAMESTATE), nl,
    play_game(GAMESTATE).
```

```
// File: game.pl
```

**Note**: Every time a new game is executed, we change the **random/3** predicate seed using an auxiliar predicate named **change_random_seed**. This way, we guarantee that the random selection is not biased and can consider various options.

To print the current board, we use **display_game/1** predicate:

```
display_game(GAMESTATE) :-
    [BOARD, SIZE, _, _,_] = GAMESTATE,
    write('    | '),
    print_columns(SIZE), nl,
    write('    '),
    print_dash_line(SIZE),
    nl,
    print_matrix(BOARD, 1).
```

```
// File: game.pl
```

### Move Validation and Execution

For the execution of the game, we use the following game cycle where the game stops when **game_over/2** predicate is activated:

```
play_game(GAMESTATE) :-
    [_,_,_, GAMEMODE,_] = GAMESTATE,
    get_move(GAMESTATE,GAMEMODE,MOVE),
    (
        move(GAMESTATE, MOVE, NEWGAMESTATE),
        display_game(NEWGAMESTATE),
```

```
        (game_over(NEWGAMESTATE, WINNER) ->
            gamewin_menu(WINNER)
        ;
            continue_game(NEWGAMESTATE)
        )
    ;
        play_game(GAMESTATE)
    ).
```

```
// File: game.pl
```

Using **get_move/3** we get a "MOVE" where we have 2 options: If the gamemode uses computer it will analise if the current player is human or computer, and if its the second case will use the **valid_moves/3** predicate to get a move. On the other hand, if its a human, the human is asked to choose a start and end coordinates. When the human inserts coordinates that are inside the board, the move/3 predicate is executed:

```
move(GAMESTATE, [START, END], NEWGAMESTATE) :-
    [BOARD,_, PLAYER, _, _] = GAMESTATE,
    valid_move(BOARD, PLAYER, [START, END], TYPE, 1),
    execute_move(GAMESTATE, [START, END], NEWGAMESTATE),
    handle_move_type(TYPE, PLAYER, [START, END]).
```

```
// File: game.pl
```

Then, the **valid_move/5** is executed to validate the move:

```
valid_move(BOARD, PLAYER, [START, END], TYPE, DISPLAYERRORS) :-
    get_piece(BOARD, START, PIECE),
    valid_piece(PLAYER, PIECE, DISPLAYERRORS),
    get_piece(BOARD, END, FPIECE),
    valid_fpiece(PLAYER, FPIECE, DISPLAYERRORS),
    get_direction(START, END, DIRECTION),
    valid_direction(DIRECTION, DISPLAYERRORS),
    get_move_linelength(BOARD, START, PIECE, DIRECTION, LINELENGTH),
    get_move_length([START, END], LENGTH),
    valid_length(LENGTH, LINELENGTH, DISPLAYERRORS),
    move_type(LENGTH, TYPE),
    valid_move_type(PLAYER, TYPE, START, DISPLAYERRORS),
    valid_position(PLAYER, END, DISPLAYERRORS).
```

```
// File: logic.pl
```

The move is considered valid when:

- The “START” position has a player piece.
- The “END” position doesn’t have a player piece.
- The direction of the move is either vertical, horizontal or diagonal.
- The move length is equal to the line length and the line length is the number of player pieces arranged as an unbroken straight line in that direction.
- The move has valid type: If the player is making continous moves, he can´t do single steps or move other pieces than the previous moved piece.
- The “END” position is not a blocked position in the player turn.

After the validation of the move, the move is executed and the board is updated. The move type is also evaluated, updating if the player can make continous moves, blocked positions and current moving piece.

```
execute_move(GAMESTATE, [START, END], NEWGAMESTATE) :-
    [BOARD, SIZE, PLAYER, GAMEMODE, BOTLEVEL] = GAMESTATE,
    get_piece(BOARD, START, PIECE),
    add_piece(BOARD, START, 0, TEMPBOARD),
    add_piece(TEMPBOARD, END, PIECE, NEWGAMEBOARD),
    NEWGAMESTATE = [NEWGAMEBOARD, SIZE, PLAYER, GAMEMODE, BOTLEVEL].


// File: logic.pl

handle_move_type(TYPE, PLAYER, [START, END]) :-
    (TYPE = single_step ->
        allow_single_steps(PLAYER)
    ;
    TYPE = jump ->
        allow_continous_moves(PLAYER),
        update_continousmove_piece(PLAYER, END),
        add_blocked_position(PLAYER, START)
    ).


// File: logic.pl
```

### List of Valid Moves

The list of valid moves uses an auxiliar built-in predicate findall(+Template, :Goal, -Bag). The “Template” in our case is “[YS-XS, YF-XF]” that represents the start coordinates and the end coordinates of the move. Then, using the player positions we try every possible combination of “[YS-XS, YF-XF]”, where YS-XS is the player position and YF-XF is the final position build using **between/3** predicate, we execute the **valid_move/5** predicate. In the end, we get a list of valid moves.

```
valid_moves(GAMESTATE, PLAYER, VALIDMOVES) :-
    [BOARD,SIZE, _, _, _] = GAMESTATE,
    get_player_positions(BOARD, PLAYER, SIZE, POSITIONS),
    findall(
        [YS-XS, YF-XF],
        (
        member([YS,XS], POSITIONS),
        between(1, SIZE, YF),
        between(1, SIZE, XF),
        valid_move(BOARD, PLAYER, [YS-XS, YF-XF],_,0)
        ),
        VALIDMOVES
    ).
```

```
// File: logic.pl
```

Since this predicate involves a wide range of option calculations, we only used this predicate for bots and to validate if the player can play again after a jump to prevent wrong answers from humans and block the game if they can't play again. For humans and single steps, we used **valid_move/5** predicate to validate its move.

### End of Game

In our game cycle, we call the **game_over/2** predicate after each move of the players, checking if any of them won the game. We first check the opponent because in our game, if both players win at the same time, the current player that made the move loses.

```
game_over(GAMESTATE, WINNER) :-
    [BOARD, SIZE, PLAYER, GAMEMODE, BOTLEVEL] = GAMESTATE,
    change_turn(PLAYER, NEXTPLAYER),
    NEWGAMESTATE = [BOARD, SIZE, NEXTPLAYER, GAMEMODE, BOTLEVEL],
    (check_win(NEWGAMESTATE) ->
        WINNER = NEXTPLAYER
    ;
    check_win(GAMESTATE) ->
        WINNER = PLAYER
    ).
```

```
// File: logic.pl
```

For the **game_over/2** predicate, we use an auxiliar predicate **check_win/1** to check if a player wins the game. The player wins the game when everyone of their pieces is positioned to avoid adjacency with their own pieces, both vertically,

horizontally, and diagonally.

```
check_win(GAMESTATE) :-
    [BOARD, SIZE, PLAYER, _,_] = GAMESTATE,
    get_player_positions(BOARD, PLAYER, SIZE, POSITIONS),
    check_positions(BOARD, PLAYER, POSITIONS).


// File: logic.pl
```

### Game State Evaluation

Each board is evaluated by the position of each piece as well as the number of pieces. For that, we use **value/3** predicate. For calculating the total board value we used the following formula: **VALUE is VALUE1 + VALUE2 + VALUE3**

- **VALUE1**: ISOLATEDPOSITIONSVALUE + (TRADE_OFF_FACTOR * ENEMYPIECES

```
ISOLATEDPOSITIONSVALUE = number of pieces from the player that are considered free by the ga
TRADE_OFF_FACTOR = 2, we consider a trade_off of 2 enemy pieces, prioritizing their existenc
ENEMYPIECES = number of pieces in the board from the other player.
```

- **VALUE2**: 1000, if the current board represents a win to the player.
- **VALUE3**: -3000, if the current board represents a win to the other player.

The **value/3** predicate is the following:

```
value(GAMESTATE, PLAYER, VALUE) :-
    [BOARD, SIZE, _, GAMEMODE, BOTLEVEL] = GAMESTATE,
    get_player_positions(BOARD, PLAYER, SIZE, POSITIONS),
    findall(
        [Y, X],
        (
            member([Y,X], POSITIONS),
            check_position(BOARD, PLAYER,[Y,X]),
            check_position_2(BOARD, PLAYER,[Y,X])
        ),
        ISOLATEDPOSITIONS
    ),
    length(ISOLATEDPOSITIONS, ISOLATEDPOSITIONSVALUE),
    length(POSITIONS, TOTALPOSITIONS),
    change_turn(PLAYER, NEXTPLAYER),
    get_player_positions(BOARD, NEXTPLAYER, SIZE, OTHERPOSITIONS),
    length(OTHERPOSITIONS, ENEMYPIECES),
    TRADE_OFF_FACTOR is 2,
    VALUE1 is (ISOLATEDPOSITIONSVALUE + (TRADE_OFF_FACTOR * ENEMYPIECES)),
    (TOTALPOSITIONS =:= ISOLATEDPOSITIONSVALUE ->
```

```
        VALUE2 = 1000
    ;
        VALUE2 = 0
    ),
    change_turn(PLAYER, NEXTPLAYER),
    NEWGAMESTATE = [BOARD, SIZE, NEXTPLAYER, GAMEMODE, BOTLEVEL],
    (check_win(NEWGAMESTATE) ->
        VALUE3 = -3000
    ;
        VALUE3 = 0
    ),
    VALUE is VALUE1 + VALUE2 + VALUE3.
```

```
// File: logic.pl
```

### Computer Plays

When the bots are playing, they have different decision type: random and greedy.

Using bot level 1, means random, we used the following predicate to get a move:

```
choose_move(GAMESTATE, PLAYER, 1, MOVE) :-
    valid_moves(GAMESTATE, PLAYER, VALIDMOVES),
    random_choice(MOVE, VALIDMOVES),
    MOVE = [YS-XS, YF-XF],
    format('Computer randomly moved from (~w-~w) to (~w-~w).', [YS, XS, YF, XF]),nl.
```

```
// File: computer.pl
```

When the random bot makes a jump and can play again, randomly chooses if wants to play again, using the following predicate:

```
get_computer_answer(GAMESTATE, _, 1) :-
    write('Since you made a jump and the jumped piece can move again, you are allowed to pla
    write('Do you want to play again (yes or no)?\n'),
    random_choice(ANSWER, ['yes', 'no']),
    format('computer randomly chose: ~w.\n', [ANSWER]),
    (
    ANSWER = 'yes' ->
        play_game(GAMESTATE)
    ;
    ANSWER = 'no' ->
        [BOARD, SIZE, PLAYER, GAMEMODE, BOTLEVEL] = GAMESTATE,
        allow_single_steps(PLAYER),
        clear_blocked_positions(PLAYER),
        remove_continousmove_piece(PLAYER),
```

```
        change_turn(PLAYER, NEXTPLAYER),
        NEWGAMESTATE = [BOARD, SIZE, NEXTPLAYER, GAMEMODE, BOTLEVEL],
        play_game(NEWGAMESTATE)
    ).
```

```
// File: computer.pl
```

On the other hand, using a bot level 2, means greedy, we combine the game state evaluation using **value/3** predicate and **minimax** algorithm to evaluate the move of the next player too and choose the best combination of current and future moves. We implemented minimax_ai/5 predicate with depth 2 that avaliates the best move of the current player taking into consideration the consequent best move of the other player. Basically, to choose the best move in a greedy way, we want to get the move that maximizes the following formula:

**value(CurrentBestMove) - value(OpponentBestMove)**

Using minimax algorithm, we maximize the move current player while minimizing the move of the next player. The move with the highest difference between moves is considered the best one.

```
choose_best_move(GAMESTATE, PLAYER, VALIDMOVES, MOVE) :-
    change_turn(PLAYER, NEXTPLAYER),
    findall(
        VALUE-MOVE,
        (
            member(MOVE, VALIDMOVES),
            execute_move(GAMESTATE, MOVE, NEWGAMESTATE),
            value(NEWGAMESTATE, PLAYER, VALUE1),
            minimax_ai(NEWGAMESTATE, NEXTPLAYER, min, 1, VALUE2),
            VALUE is VALUE1 + VALUE2
        ),
        MOVESVALUE
    ),
    keysort(MOVESVALUE, SORTEDMOVESVALUE),
    last(SORTEDMOVESVALUE, MAX-_),
    findall(MOVES, member(MAX-MOVES, SORTEDMOVESVALUE), MAXMOVES),
    random_choice(MOVE, MAXMOVES).
```

```
// File: computer.pl
```

```
minimax_ai(_, _, _, 2, 0):- !.
minimax_ai(GAMESTATE, PLAYER, MODE, DEPTH, VALUE):-
    change_turn(PLAYER, NEXTPLAYER),
    minimax_mode(MODE, NEWMODE),
    NEWDEPTH is DEPTH + 1,
```

```
    valid_moves(GAMESTATE, PLAYER, VALIDMOVES),
    findall(TEMPVALUE,
            (member(MOVE, VALIDMOVES),
            execute_move(GAMESTATE, MOVE, NEWGAMESTATE),
            value(NEWGAMESTATE, PLAYER, VALUE1),
            minimax_ai(NEWGAMESTATE, NEXTPLAYER, NEWMODE, NEWDEPTH, VALUE2),
            TEMPVALUE is VALUE1 + VALUE2
            ),
        VALUES),
    sort(VALUES, SORTEDVALUES),
    minimax_value(MODE, SORTEDVALUES, VALUE).
```

```
// File: computer.pl
```

If many moves have the same highest value, the bot randomly chooses one in order to avoid infinite cycles.

## Conclusions

The *Apart* game has been successfully implemented in Prolog, with the possibility of choosing a board size between 8 and 25. Game modes include human vs human, human vs computer, computer vs human and computer vs computer. When playing against the computer, there are two bots of different difficulty. All moves are validated to ensure that the game works correctly.

During the development of the project, we applied the knowledge acquired during the semester, resulting in the successful implementation of the planned functionalities. However, the most challenging aspect was the implementation of the greedy bot, since it has to evaluate every possible move, always using the best one.

## Bibliography

We consulted the following websites to see the description and rules of the game.

https://kanare-abstract.com/en/pages/apart

https://cdn.shopify.com/s/files/1/0578/3502/8664/files/Apart_EN.pdf?v=1682248406