CONTROL + V

Programação em Lógica

Relatório Final

17 de novembro de 2019

Bernardo Manuel Esteves dos Santos up201706534@fe.up.pt
Vítor Hugo Leite Gonçalves up201703917@fe.up.pt

Resumo

No âmbito da unidade curricular **Programação em Lógica** foi-nos proposto a conceção de um jogo de tabuleiro utilizando a linguagem de programação **Prolog**.

O jogo desenvolvido chama-se **Control + V** e tem como principal objetivo a cópia de peças já no tabuleiro para novas posições, mantendo as anteriores, seguindo certas regras de simetria.

Inicialmente foram encontradas algumas dificuldades relacionadas com o novo paradigma que nos foi apresentado por esta nova linguagem de programação bastante diferente das que estamos habituados, mas após estas serem ultrapassadas a implementação de toda a lógica do jogo fluiu naturalmente, tornando o desenvolvimento do mesmo bastante divertido ainda que desafiante.

No fim da implementação do jogo, pudemos concluir que conseguimos com sucesso aplicar os conceitos abordados ao longo das aulas, bem como a elaboração de um jogo simples e de fácil interação.

Em suma, podemos afirmar que após a conclusão do projeto os nossos conhecimentos sobre a linguagem aumentaram significativamente e estamos bastantes orgulhosos do produto final.

Índice

1. Introdução

O projeto tem como objetivo a aplicação do conhecimento adquirido e abordado durante as aulas teóricas e práticas da unidade curricular Programação em Lógica do 1º semestre do 3º ano do Mestrado Integrado em Engenharia Informática e de Computação, tendo como foco o jogo de tabuleiro Control + V.

O alvo deste relatório passa pela exposição e esclarecimento de qualquer dúvida que possa ter surgido, descrevendo a lógica do jogo, focando essencialmente da representação do estado de jogo, visualização, regras de movimentação, condições de terminação de jogo, entre outros.

O jogo permite 4 modos de jogo distintos, sendo eles Player vs Player, Player vs Computer, Computer vs Player, Computer vs Computer, sendo que o jogador Computer tem dois níveis distintos, o nível 'random' e o 'greedy' que serão explicados na secção Jogadas do Computador.

O relatório está dividido da seguinte forma:

- **O jogo Control + V :** Descrição do jogo e explicação das regras do mesmo.
- **Lógica do jogo :**
  - Representação do estado de jogo: exemplificação dos vários estados que podem ocorrer ao longo do jogo.
  - Visualização do tabuleiro em modo texto: apresentação do conjunto de predicados que permite a visualização do mesmo.
  - Interface do jogo: apresentação da interface apresentada ao jogador.
  - Lista de jogadas válidas: predicados usados para a validação das jogadas.
  - Execução de jogadas: ciclo de jogo e a forma como é efectuada cada jogada.
  - Final do jogo: predicados que verificam quando o jogo chega ao final.
  - Avaliação do tabuleiro: predicado que faz a avaliação do estado do jogo.
  - Jogadas do computador: predicados utilizados pelo computador para efetuar jogadas.

2. O jogo Control + V

2.1. Descrição

Control + V é um jogo de estratégia abstrata no qual o jogador tem como objetivo ocupar a maior área de tabuleiro possível com os blocos da sua cor. Para atingir este objetivo o jogador pode copiar (ctrl+c) os blocos que já se encontram colocados no tabuleiro e colá-los (ctrl+v) numa nova posição, através de rotações em torno de pontos ou simetrias, respeitando as regras do jogo.

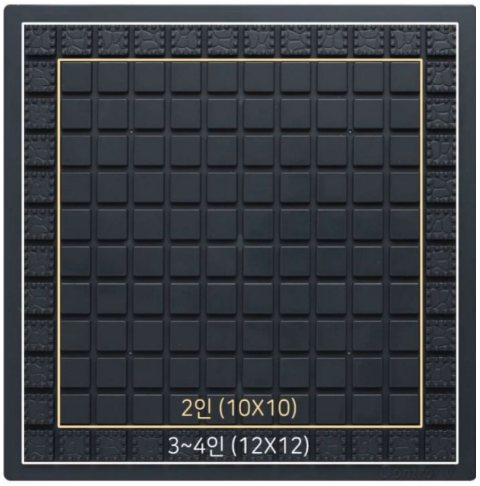
O tamanho do tabuleiro é dependente do número de jogadores, sendo 10x10 para 2 jogadores e 12x12 para 3 ou 4 jogadores.

2.2. Regras

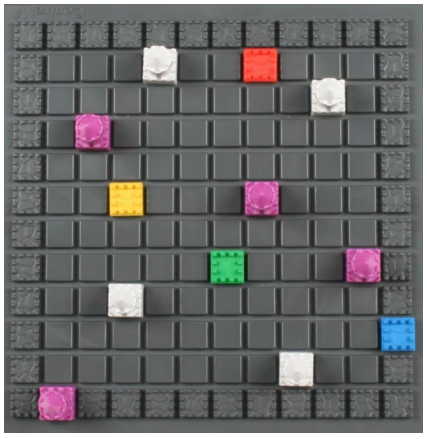
Originalmente, o jogo está feito para ser jogado por 2 a 4 pessoas. No entanto, para esta unidade curricular, iremos **reduzir o número de jogadores para 2**. É importante referir que, independentemente do número de jogadores, as regras mantêm-se inalteradas.

Existem **dois tipos de peças**:

- as que representam o **território** de cada jogador, tendo cada jogador peças de uma só cor;
- os **castelos**, 2 por jogador, cujo objetivo é serem obstáculos presentes no tabuleiro, de forma a aumentar a complexidade das jogadas.

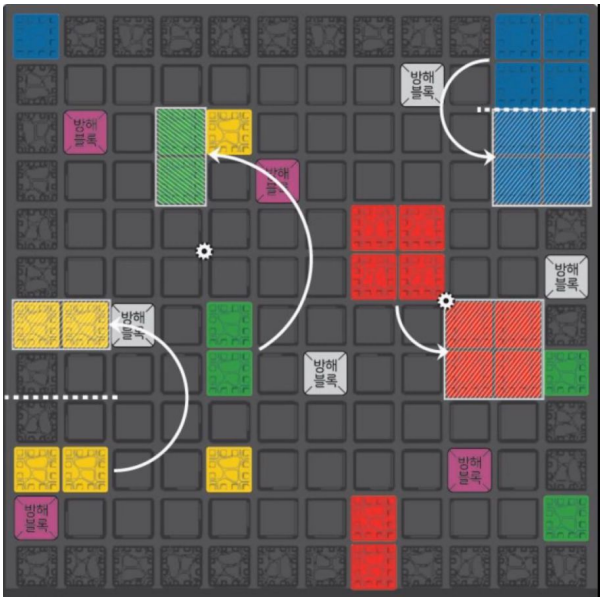


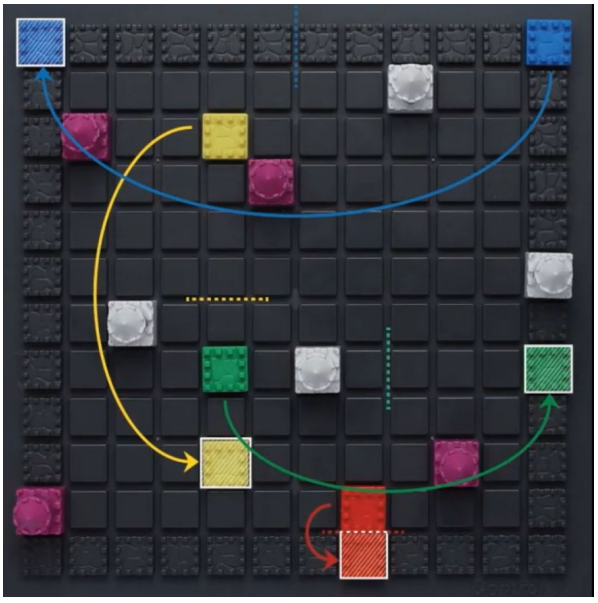
O jogo começa com um tabuleiro **10x10** totalmente **vazio** (interior do quadrado de contorno amarelo representado à direita).



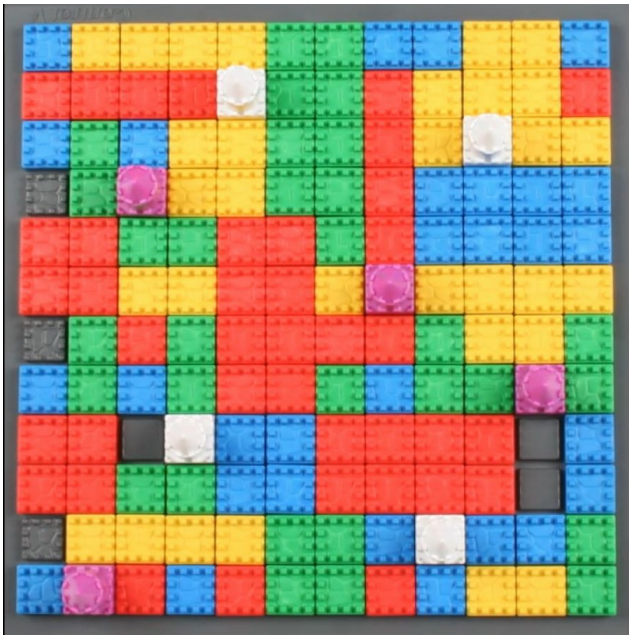
No início do jogo, cada um dos jogadores posiciona **dois castelos em duas casas do tabuleiro à sua escolha**. Depois de todos os castelos estarem em posição, todos os jogadores colocam um dos seus blocos coloridos na posição que considerarem mais adequada (exemplo do lado esquerdo).

A partir deste momento, e até ao final do jogo, cada jogador, alternadamente, terá de copiar um dos seus blocos de peças (peças pertencentes ao jogador que se encontrem ligadas ou uma peça do jogador que se encontre isolada) e colá-lo numa nova posição. A colagem do bloco apenas pode ser realizada através de uma **simetria segundo os eixos axiais(x, y)** ou segundo uma **rotação em torno de um determinado ponto (interseção de dois eixos)** do tabuleiro. Para ser possível a colagem o espaço que será ocupado pelo bloco tem se estar completamente livre. Nas figuras abaixo estão representados alguns exemplos de possíveis jogadas.





O jogo termina quando nenhum dos jogadores conseguir colocar mais peças no tabuleiro. Na figura abaixo encontra-se um possível estado final para o tabuleiro num jogo com 4 jogadores.



3. Lógica do jogo

3.1. Representação do Estado do Jogo

O **tabuleiro de jogo** é representado em prolog por uma **lista de listas**, de modo a facilitar a sua representação e a visualização do mesmo. Cada célula é um número, sendo que o número 0 corresponde a uma célula vazia, 1 é uma peça do jogador 1, 2 é uma peça do jogador 2 e 3 é um castelo.

Pode ser observado nas figuras da próxima seção a representação interna de alguns possíveis estados de jogo.

3.1.1. Estado Inicial

```
initialBoard([
    [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, 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, 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]]).
```

3.1.2. Estado Intermédio

```
intermidiateBoard([
    [0, 0, 1, 0, 2, 1, 0, 2, 0, 3],
    [0, 0, 0, 0, 0, 0, 0, 0, 0, 0],
    [0, 0, 0, 0, 0, 0, 0, 0, 0, 0],
    [0, 0, 1, 1, 2, 3, 0, 0, 0, 0],
    [0, 0, 1, 1, 0, 0, 0, 0, 0, 0],
    [0, 0, 0, 0, 0, 0, 2, 2, 0, 0],
    [0, 0, 0, 3, 0, 0, 0, 3, 0, 0],
    [0, 0, 1, 1, 1, 1, 0, 0, 0, 0],
    [0, 0, 0, 0, 0, 0, 2, 2, 0, 0],
    [0, 0, 0, 0, 0, 0, 2, 2, 0, 0]]).
```

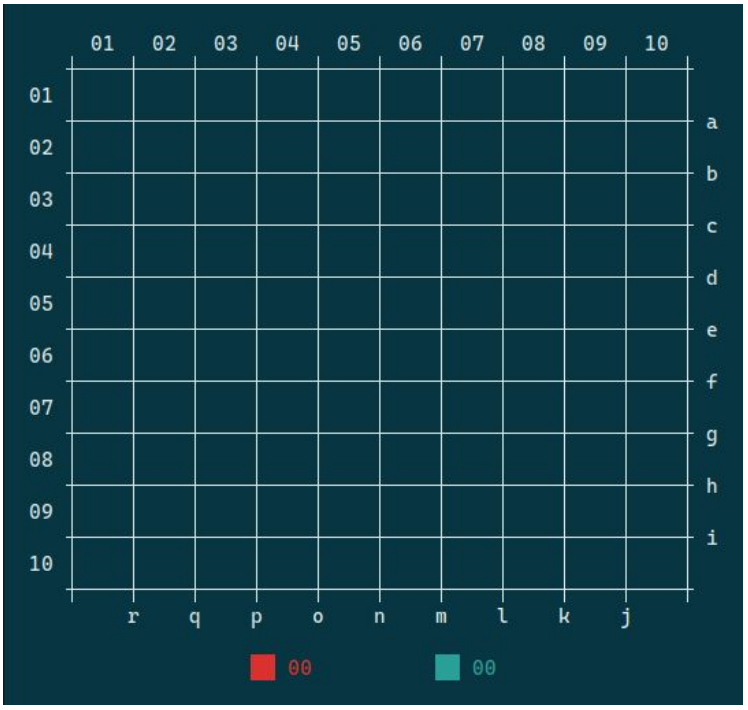
3.1.3. Estado Final

```
finalBoard([
    [0, 1, 1, 0, 2, 1, 1, 2, 2, 3],
    [0, 1, 1, 0, 2, 1, 1, 2, 2, 0],
    [0, 0, 2, 2, 2, 1, 1, 1, 1, 0],
    [1, 1, 1, 1, 2, 3, 0, 2, 2, 0],
    [1, 1, 1, 1, 0, 0, 0, 2, 2, 0],
    [0, 0, 2, 2, 2, 2, 2, 2, 0, 0],
    [2, 2, 0, 3, 0, 0, 0, 3, 0, 1],
    [2, 2, 1, 1, 1, 1, 1, 1, 1, 1],
    [2, 2, 2, 2, 0, 0, 2, 2, 2, 2],
    [2, 2, 2, 2, 0, 0, 2, 2, 2, 2]]).
```

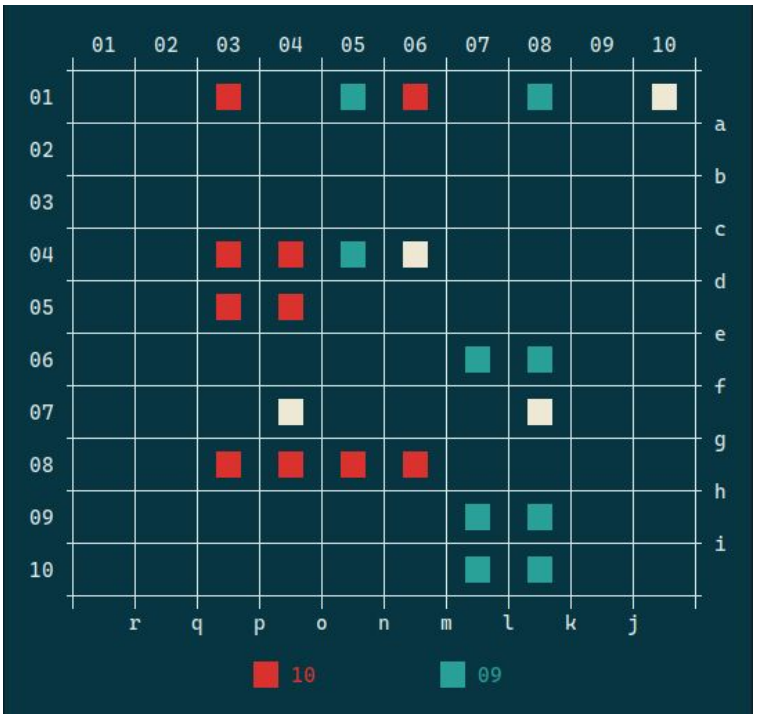
3.2. Visualização do tabuleiro em modo texto

Abaixo encontra-se a representação do tabuleiro em modo de texto, os blocos são representados por 2 espaços coloridos conforme o tipo de célula. As linhas e colunas identificadas por números são utilizados para representar a posição das peças. Os eixos que estão identificados por letras servem para realizar as simetrias.

3.2.1. Estado Inicial



3.2.2. Estado Intermédio



3.2.3. Estado Final

3.3. Lista de jogadas válidas

Para que uma **jogada seja válida**, neste caso consideramos uma jogada a seleção do(s) bloco(s) a mover e também do tipo de simetria e dos eixos da mesma, é necessário que **todas as posições das novas peças** coincidam com **células vazias**.

Através do predicado ***valid_moves,*** e através da informação dos seus átomos é possível descobrirmos **todas as jogadas válidas** que um jogador pode efetuar **naquele momento** de jogo.

```
% Get the list of all valid moves for a player
valid_moves(Board, Piece, ListOfMoves) :-
    findall(Move, validMove(Board, Piece, _Symmetry, _Row, _Column, _Axes, _, Move), ListOfMoves),
    length(ListOfMoves, Length),
    Length > 0.
```

ListOfMoves é uma lista com todos os Move que satisfazem o predicado ***validMove***. O predicado só retorna verdade se a lista tiver elementos, ou seja, o seu tamanho for **maior** que 0.

Com o predicado ***validMove,*** conseguimos através de predicados auxiliares como **between**, ***getBlockPositions*** e ***symmetry***, gerar todas as combinações válidas de jogadas de um jogador.

Para melhor compreensão dos predicados utilizados para a obtenção da lista de jogadas válidas, segue uma descrição sucinta de dois predicados.

- **getBlockPositions:** permite, através de uma célula, identificada por Row e Column, retornar uma lista de posições de todas as células adjacentes a esta, que na lógica do jogo funcionam como uma só.
- **symmetry:** permite obter a lista das posições das novas peças, caso a simetria escolhida seja válida.

```
% Check if a move is valid and return the positions of the new blocks
validMove(Board, Piece, Symmetry, Row, Column, Axes, NewPositions, ValidMove) :-
    between(0, 9, Row),
    between(0, 9, Column),
    between(0, 2, Symmetry),
    getBlockPositions(Board, Row, Column, Piece, Positions),
    symmetry(Board, Positions, Symmetry, Axes, NewPositions),
    ValidMove = Row-Column-Symmetry-Axes.
```

3.4. Execução de jogadas

O jogo tem um predicado que é o principal responsável pelo ciclo de jogo e pela verificação de terminação do mesmo, o ***gameLoop.***

```
% When the game is over print the winner
gameLoop(Board, _Player1, _P1Level, _Player2, _P2Level) :-
    game_over(Board, Winner),
    printWinner(Winner).
% Loop during which the players make their moves, one move for each.
% When a player has no moves left he passes the turn to the other player
gameLoop(Board, Player1, P1Level, Player2, P2Level) :-
    playerTurn(Board, Player1, P1Level, 1, NewBoard1),
    playerTurn(NewBoard1, Player2, P2Level, 2, NewBoard2),
    % if there are no moves left game_over, else keep playing
    gameLoop(NewBoard2, Player1, P1Level, Player2, P2Level).
```

É dentro do predicado ***playerTurn*** que se desenvolve toda a jogada relativa ao jogador. Para efetuar uma jogada completa, o jogador terá de indicar primeiramente as coordenadas do bloco que pretende copiar aquando da chamada do predicado ***getBlock***. Após a verificação que este pertence ao jogador em questão é pedido ao jogador que dê informações sobre a simetria a efetuar, usamos um predicado ***askSymmetry***. É chamado então o predicado ***move***, que recebe nos seus átomos informação sobre o **estado do Board**, o tipo de **peça**, e um **tuplo** que permite saber todas a informação relativa à jogada a efetuar.

```
% Player turn
playerTurn(Board, 'P', _Level, Piece, NewBoard) :-
    % check if the player has any moves left
    valid_moves(Board, Piece, _ValidMoves),
    % get a block to make the symmetry
    getBlock(Board, Piece, Row, Column),
    % ask for the symmetry
    askSymmetry(Symmetry, Axes),
    Move = Row-Column-Symmetry-Axes,
    % make the move if possible
    move(Board, Move, Piece, NewBoard),
    printMove(Move, Piece),
    printBoard(NewBoard).
```

Este predicado tem como objetivo validar o movimento a ser efetuado pela simetria, chamando um predicado já visto anteriormente, ***validMove***, que retorna uma lista de **novas posições**. Caso seja possível efetuar a jogada, é chamado um outro predicado ***updateBoard***, que utilizando o Board passado como átomo, retorna um novo Board atualizado.

```
% Check if a move is valid and, if it is, execute it
move(Board, Move, Piece, NewBoard) :-
    Row-Column-Symmetry-Axes = Move,
    validMove(Board, Piece, Symmetry, Row, Column, Axes, Pos, _),
    updateBoard(Board, Pos, Piece, NewBoard).
```

3.5. Final do jogo

Tal como enunciado nas **regras do jogo**, este acaba quando **nenhum dos jogadores** tiver mais jogadas válidas a efetuar. Para se encontrar o vencedor basta no final do jogo, **calcular o número de peças** de cada um no tabuleiro, e aquele que tenha o maior número é o vencedor. Caso o número de peças seja igual é considerado um **empate**.

Para a implementação lógica de terminação de jogo, foi necessário a cada iteração do game loop verificar o predicado ***game_over***, que nos diz se ainda há ou não jogadas válidas por parte de algum dos jogadores. Se houver, o game loop prossegue normalmente, em caso contrário é analisado o estado final do tabuleiro, calculando o número de peças de cada jogador e que nos permite saber quem é o vencedor, ou se houve empate.

1. ***game_over***
   Chama o predicado ***valid_moves*** para cada um dos jogadores, e caso um deles ainda tenha jogadas válidas, o predicado retorna falso. Caso contrário, é calculado o número de peças em jogo de cada um e chamado o predicado *getWinner*.

```
game_over(Board, Winner):-
    \+valid_moves(Board, 1, _),
    \+valid_moves(Board, 2, _),
    !,
    value(Board, 1, V1),
    value(Board, 2, V2),
    getWinner(V1, V2, Winner).
```

2. ***getWinner***
   Recebe o número de peças de cada jogador, compara os valores e devolve o vencedor (0 no caso de empate, o número do jogador quando há um vencedor).

```
getWinner(Points1, Points2, 1) :-
    Points1 > Points2.
getWinner(Points1, Points2, 2) :-
    Points2 > Points1.
getWinner(_Points1, _Points2, 0).
```

3. ***printWinner***
   Recebe um átomo com o jogador vencedor e imprime o nome do player. Caso haja empate retorna *'draw'.*

```
% Print the result of the game(draw, player 1 won, player 2 won)
printWinner(0) :-
    ansi_format([bg(black), fg(red)], '  ┼──────────────────────────────────────────┼  ', []), nl,
    ansi_format([bg(black), fg(red)], '  |                It\'s a draw...           |  ', []), nl,
    ansi_format([bg(black), fg(red)], '  ┼──────────────────────────────────────────┼  ', []), nl.
printWinner(Winner) :-
    ansi_format([bg(black), fg(red)], '  ┼──────────────────────────────────────────┼  ', []), nl,
    ansi_format([bg(black), fg(red)], '  |                PLAYER ~w WON             |  ', [Winner]), nl
    ansi_format([bg(black), fg(red)], '  ┼──────────────────────────────────────────┼  ', []), nl.
```

3.6. Avaliação do tabuleiro

De acordo com as regras do jogo, a pontuação de cada jogador é correspondente ao número de peças que tem colocadas no tabuleiro. O predicado ***value*** realiza essa contagem na ***Board*** que lhe é passada para o ***Player*** passado, e retorna a contagem através do termo ***Value***.

```
% Get the score of a player from the current
value(Board, Player, Value):-
    countItemsMatrix(Board,Player,Value).
```

Foram desenvolvidos outros dois predicados de modo a que fosse possível fazer essa contagem. O predicado ***countItemsMatrix*** conta o número de vezes que um elemento se repete numa matriz, e o predicado ***countItemsList*** conta o número de vezes que um elemento se repete numa lista.

```
% counts how many times an item appears at a matrix
countItemsMatrix([], _Item, 0).
countItemsMatrix([H|T], Item, Count):-
    countItemsList(H, Item, Count2),
    countItemsMatrix(T, Item, Count1),
    Count is Count1 + Count2.
```

```
% counts how many times an item appears at a list
countItemsList([], _Item, 0).
countItemsList([H|T], Item, Count) :-
    Item == H,
    !,
    countItemsList(T, Item, Count1),
    Count is Count1 + 1.
countItemsList([_|T], Item, Count) :-
    countItemsList(T, Item, Count).
```

3.7. Jogadas do computador

Como já foi referido anteriormente, existem alguns modos neste jogo que nos permitem jogar contra o computador, podendo este jogar com duas estratégias diferentes. Um primeiro random, que escolhe jogadas válidas aleatórias e um que analisa as jogadas válidas que pode efetuar escolhendo a melhor opção no momento, sendo esta a opção que coloca o maior número de peças no tabuleiro.

No início do jogo o computador coloca os castelos e a sua peça inicial no tabuleiro em coordenadas aleatórias geradas pelo predicado ***generateCoords***.

```
generateCoords(Board, Row, Column) :-
    random(0, 10, Row),
    random(0, 10, Column),
    isEmpty(Board, Row, Column).
generateCoords(Board, Row, Column) :-
    generateCoords(Board, Row, Column).
```

O predicado startGame recebe sempre 4 átomos, independentemente do modo de jogo escolhido, sendo dois deles onde são especificados os tipos de jogador, podendo ser 'P' ou 'C'. Os outros dois átomos são os níveis do jogador, que apenas nos interessam no caso deste ser do tipo 'C'.

O predicado ***startGame*** passa os seus quatro átomos ao predicado ***game_loop***, e neste é passado ainda o tabuleiro atual, **Board.**

A partir daqui desenrola-se tudo idêntico a uma jogada de um jogador **'P',** no entanto como um jogador **'C'** faz as suas jogadas automaticamente sem nenhuma interferência do utilizador há predicados exclusivos para este tipo de jogador.

```
% Computer turn
playerTurn(Board, 'C', Level, Piece, NewBoard) :-
    % choose the best move, according to the level
    choose_move(Board, Piece, Level, Move),
    move(Board, Move, Piece, NewBoard),
    printMove(Move, Piece),
    printBoard(NewBoard).
```

O predicado ***choose_move*** é responsável por escolher o melhor movimento a realizar pelo computador de acordo com o nível do mesmo. Abaixo está o predicado utilizado pelo computador de nível 1, o computador vai buscar todas as jogadas possíveis e escolhe uma jogada aleatória.

```
% Choose the move for the level 1 computer
choose_move(Board, Piece, 1, Move) :-
    % get the list of valid moves
    valid_moves(Board, Piece, ValidMoves),
    % choose a random move from the list
    random_member(Move, ValidMoves).
```

Para o computador de nível 2, o predicado ***choose_move*** funciona de um modo um pouco diferente. A jogada não é escolhida de forma aleatória, o predicado ***getBestMove*** itera sobre todos os elementos da lista de jogadas válidas e escolhe a jogada que coloca mais peças no tabuleiro nessa mesma jogada.

```
choose_move(Board, Piece, 2, Move) :-
    % get the list of valid moves
    valid_moves(Board, Piece, ValidMoves),
    % get the move that places the most pieces on the board
    getBestMove(Board, Piece, ValidMoves, Move).
```

```
getBestMove(_Board, _Piece, [BMove], BMove) :- !.
getBestMove(Board, Piece, [HMove|TMove], BMove) :-
    getMoveValue(Board, Piece, HMove, HValue),
    getBestMove(Board, Piece, TMove, BMove),
    getMoveValue(Board, Piece, BMove, BValue),
    BValue ≥ HValue,
    !.
getBestMove(_, _, [HMove|_], HMove) :- !.
```

4. Conclusão

Apesar das dificuldades encontradas no início do projeto estas foram desaparecendo com o decorrer do mesmo.

Não obstante, ficamos um pouco aquém das nossas expectativas, dado que não foi possível, devido a questões de tempo, a implementação de um terceiro nível de inteligência artificial mais inteligente que o 'greedy', apesar deste não ter sido pedido.

O trabalho que agora se conclui exigiu um grande esforço de ambos os elementos do grupo, mas no final achamos que foi uma experiência diferente mas ao mesmo tempo bastante enriquecedora e satisfatória.

Em suma, estamos extremamente contentes e satisfeitos com o resultado final apresentado, tendo este contribuindo bastante para a nossa integração neste mundo das linguagens de programação declarativas.

5. Referências

- https://boardgamegeek.com/boardgame/283848/control-v
- https://www.youtube.com/watch?v=t615ediK8EY

6. Observações

O projeto foi desenvolvido usando o ***Swi Prolog.***