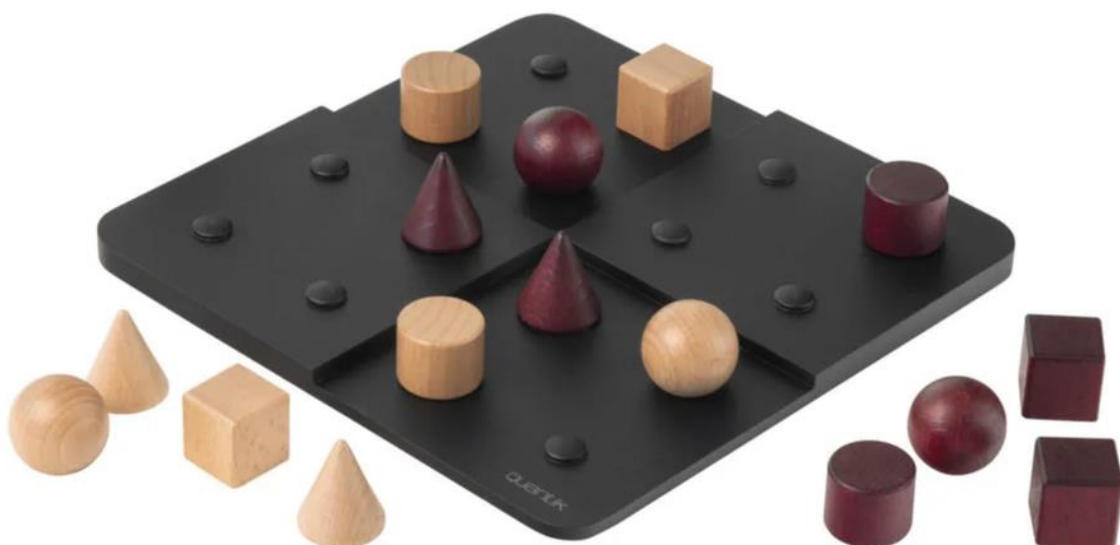


Quantik

Relatório final realizado no âmbito da disciplina de programação em lógica que integra o terceiro ano do Mestrado Integrado em Engenharia Informática e Computação na FEUP



Grupo Quantik3, constituído pelos elementos:

Gaspar Santos Pinheiro

up201704700@fe.up.pt

Maria Helena Viegas Oliveira Ferreira

up201704508@fe.up.pt

Índice

Índice	1
Introdução	2
O Jogo - Quantik	2
Tabuleiro	2
Peças	3
Objetivo	3
Regras	3
Lógica do Jogo	4
Representação Interna do Estado do Jogo	4
Visualização do tabuleiro	9
Lista de jogadas válidas	9
Execução de jogadas	10
Final do jogo	13
Avaliação do tabuleiro	13
Jogada do computador	15
Conclusões	16
Bibliografia	17

Introdução

Este trabalho tem como objetivo implementar, em linguagem Prolog, um jogo de tabuleiro para dois jogadores que permita três modos de jogo: Humano contra Humano, Humano contra Computador e Computador contra Computador. Foram também pedidos pelo menos dois níveis de dificuldade para o computador e uma interface adequada com o utilizador, em modo de texto. O sistema de desenvolvimento utilizado foi o SICStus Prolog, que inclui a possibilidade de criação de sockets para a comunicação com o módulo de visualização.

O Jogo - Quantik

Quantik é um jogo de tabuleiro de estratégia puramente abstrato para dois jogadores. Este jogo foi desenhado por Nouri Khalifat e lançado no ano de 2019 pela desenvolvedora Gigamic.

Tabuleiro

O jogo é decorre num tabuleiro dividido em 4 zonas iguais, cada uma com 2 linhas e 2 colunas, num total de 16 células.



Figura 1 - Tabuleiro vazio.

Peças

Cada jogador tem em sua posse 8 peças de 4 formas diferentes: 2 cubos, 2 cilindros, 2 esferas e 2 cones da sua cor. Sendo as peças do jogador 1 de cor branca e as do jogador 2 de cor castanha.



Figura 2 - Imagem ilustrativas das peças do jogo.

Objetivo

O objetivo do jogo é ser o primeiro jogador a ocupar uma linha, coluna ou zona quadrada com as quatro diferentes peças: o cilindro, a esfera, o cubo e o cone. As peças que formam a linha, coluna ou zona vencedora não têm que ser todas suas, poderão ser do outro jogador também.

Regras

Em cada jogada, os jogadores colocam de forma alternada uma das suas peças no tabuleiro.

Ao colocar uma peça no tabuleiro, cada jogador deve respeitar a seguinte regra: não é permitido colocar uma peça numa linha, coluna ou quadrado extremidade do tabuleiro na qual o adversário já tenha colocado uma peça da mesma forma. Um quadrado extremidade do tabuleiro é um dos quatro quadrados formados nas extremidades do tabuleiro. Sendo um exemplo de um quadrado o formado pelas 4 posições (1,1), (1,2), (2,1) e (2,2), normalmente designado no nosso código por quadrado um.

Lógica do Jogo

O predicado de início de jogo é `play/0`, que nos leva ao main menu mostrado abaixo. Este menu dá-nos seis opções, entre elas jogar (contra outra pessoa, o computador ou ver dois computadores a jogarem), ver as regras do jogo e parar a execução do jogo.

```

                                Welcome to Quantik!!!!

Your Options:

    1. Player vs Player
    2. Player vs Computer
    3. Computer vs Player
    4. Computer vs Computer
    5. help
    0. Exit

-----
Made by:
        Gaspar Santos Pinheiro
        Maria Helena Ferreira

> What is your option ? █
```

Figura 3 - Main menu do jogo

A exceção do predicado `play` todas as iterações entre jogador e o jogo podem ser feitas sem o ponto final.

Representação Interna do Estado do Jogo

Usamos uma lista de listas para representar o tabuleiro. Usando átomos para representar o estado de uma célula, todos eles numéricos para facilitar verificações e jogadas do computador:

- 0 - não ocupada;
- 1<nº do jogador> - ocupada por um cone;
- 5<nº do jogador> - ocupada por um cubo.
- 7<nº do jogador> - ocupada por um cilindro;
- 9<nº do jogador>- ocupada por uma esfera;

Onde está a tag <nº do jogador> deverá estar 1 ou 2, consoante se trate da peça do jogador 1 (de cor branca) ou do jogador 2 (de cor castanha).

Estado Inicial

```
init_board([
    [0, 0, 0, 0],
    [0, 0, 0, 0],
    [0, 0, 0, 0],
    [0, 0, 0, 0]
```

No SICStus, o resultado da visualização do tabuleiro seria este:

)).


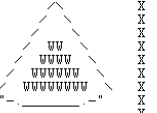

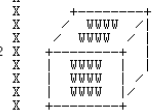
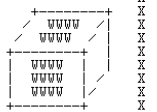
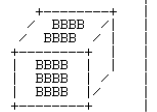



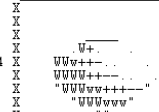
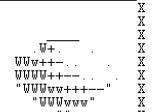
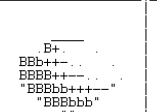
WHITE PIECES		1	2	3	4	BROWN PIECES	
1						1	
2						2	
3						3	
4						4	
Player 1, it's your turn! Your pieces are colored white What piece do you want to play? : ■							

Figura 4 - Representação do tabuleiro vazio no SICStus

Estado Intermediário

mid_board ([
 [91,0, 0, 11],
 [52, 0, 0, 0],
 [0, 51, 12, 0],
 [0, 0, 0, 52]

]).

No SICStus, o resultado da visualização do tabuleiro seria este:

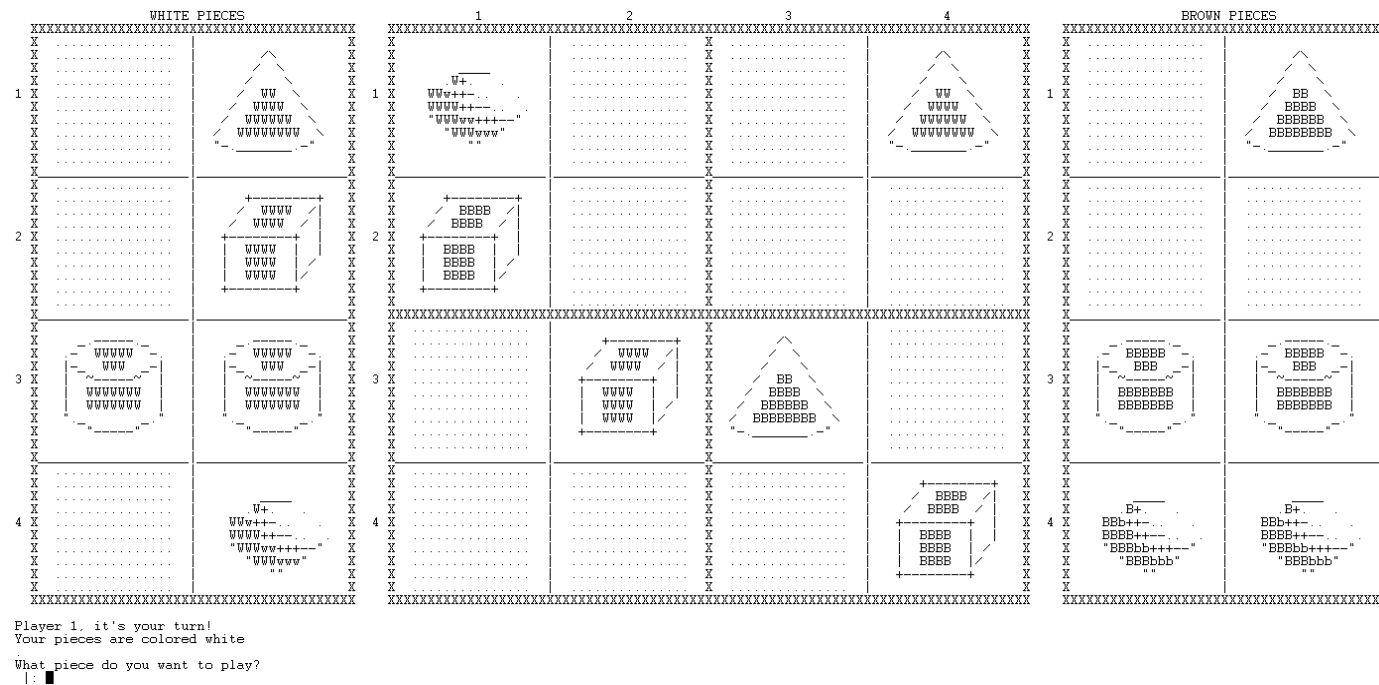


Figura 5 - Representação do tabuleiro num estado intermédio no SICStus

Nota: O próximo jogador seria o que possui as peças brancas.

Estado empate

```
end_board ([
  [91, 0, 0, 11],
  [52, 71, 92, 11],
  [ 0, 51, 12, 92],
  [ 0, 0, 0, 52]
```

]).

No SICStus, o resultado da visualização do tabuleiro seria este:

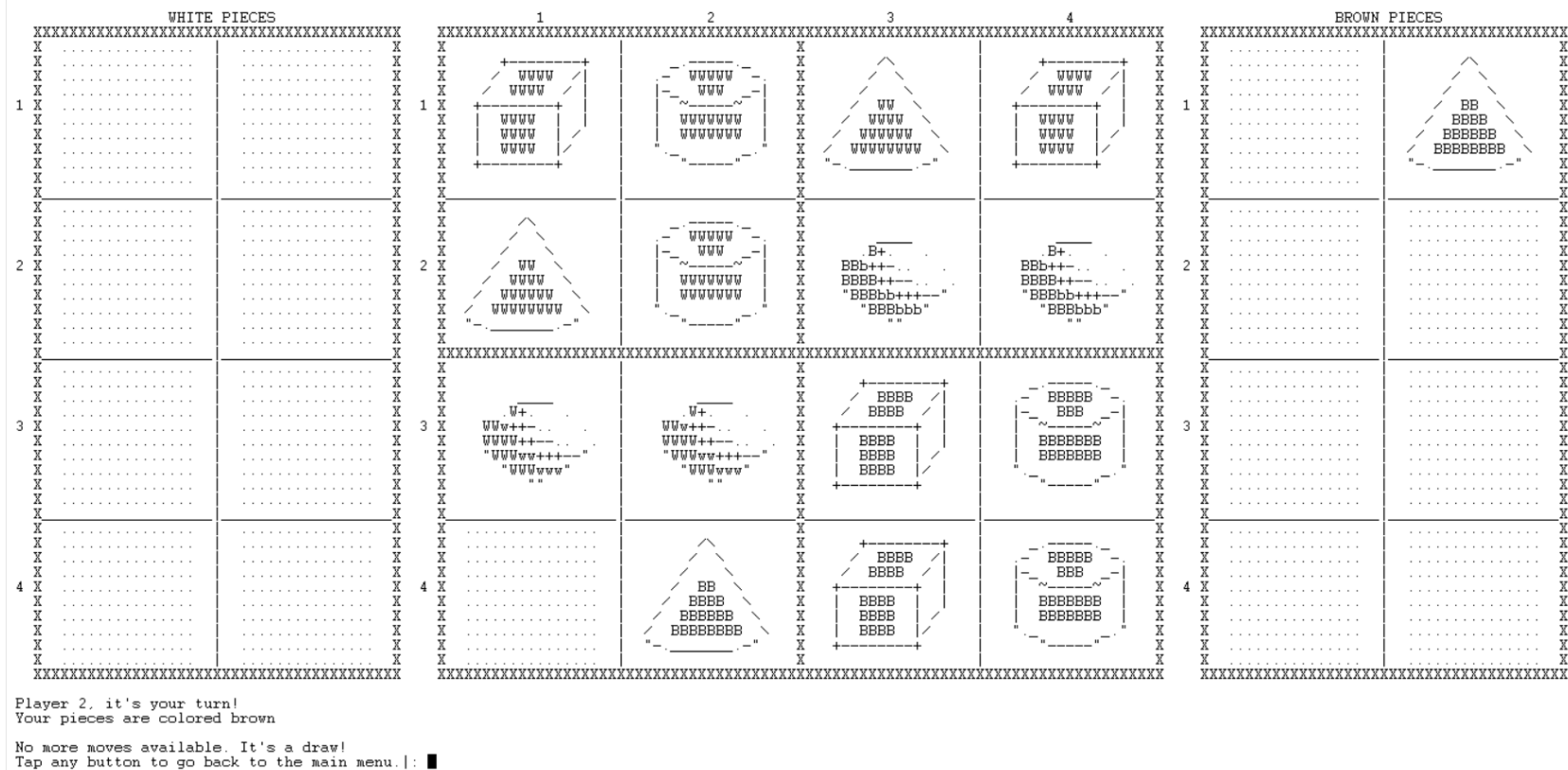
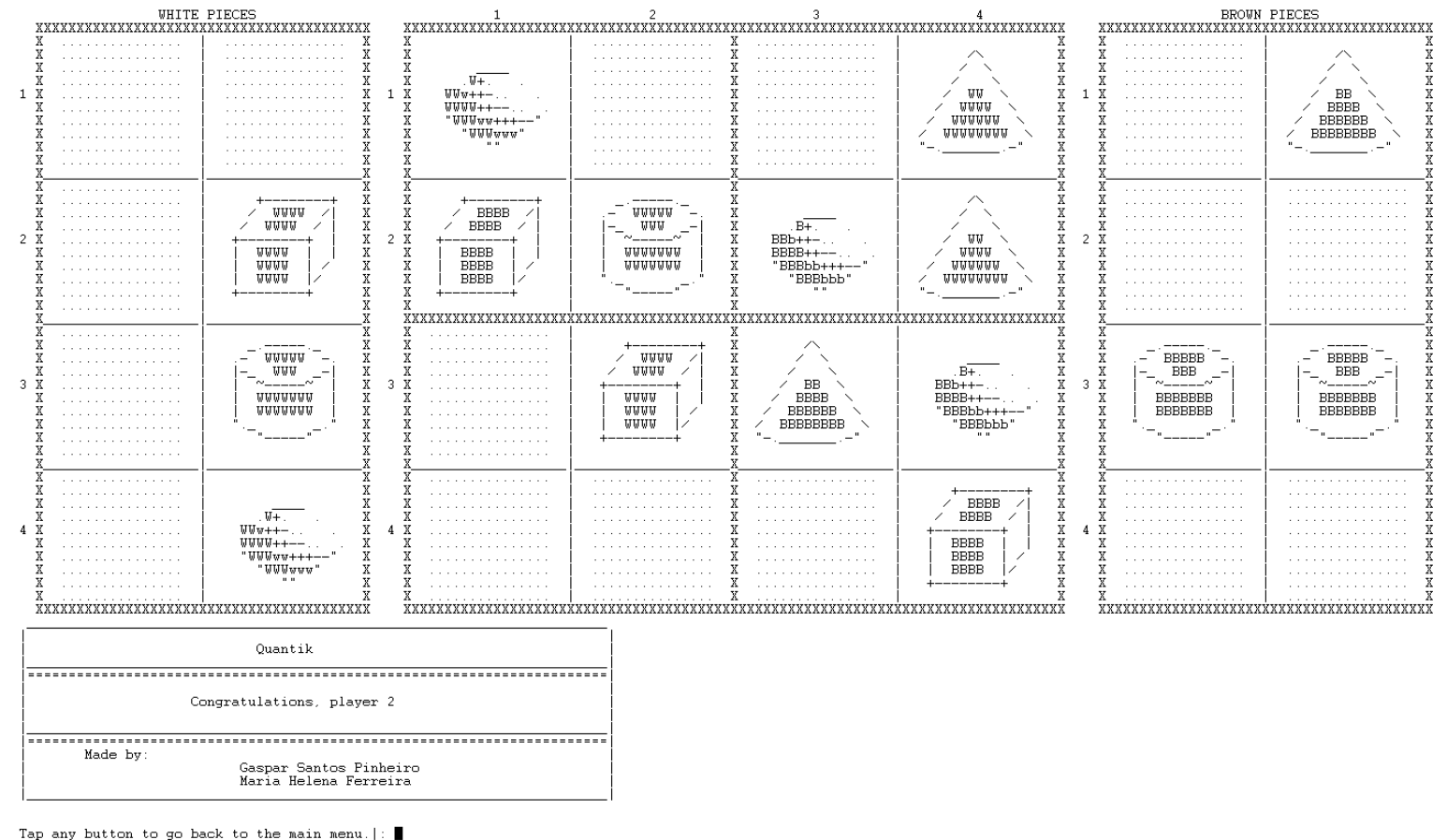


Figura 5 - Representação do tabuleiro num estado intermédio no SICStus


```
end_board ([
    [91, 0, 0, 11],
    [52, 71, 92, 11],
    [ 0, 51, 12, 92],
    [ 0, 0, 0, 52]
]).
```

No SICStus, o resultado da visualização do tabuleiro seria este:



Visualização do tabuleiro

```
4. display_game(Board, Player, White_Pieces, Brown_Pieces) :-  
5.   clear_screen,  
6.   print_header_line,  
7.   print_boards_content(Board, White_Pieces, Brown_Pieces, 1),  
8.   greet_player(Player).
```

Figura 7 - Código para visualização do tabuleiro (Ficheiro: view/board_printer.pl)

O predicado `display_game` tem como argumentos o tabuleiro a ser visualizado, o jogador que irá realizar a próxima jogada e as peças disponíveis dos dois jogadores (tanto brancas, como castanhas), pois o outro jogador poderá achar relevante saber as peças disponíveis do adversário para realizar a sua jogada.

Lista de jogadas válidas

Nível 1

```
8. valid_moves(1, Board, White_Pieces, Brown_Pieces, Player, List_Of_Moves) :-  
9.   getPiecesAvailable( White_Pieces, Brown_Pieces, New_White_Pieces, New_Brown_Pieces),  
10.  setof(Move, valid_move(0, Move, Player, Board, New_White_Pieces, New_Brown_Pieces), List_Of_Moves).
```

Figura 8 - Código para obter todas as jogadas possíveis atuais no nível 1 (Ficheiro: controller/computer.pl)

Quando o jogador escolhe o primeiro nível, apenas são escolhidas jogadas aleatórias (dentro das jogadas válidas). O número de jogadas possíveis pode ser calculado da seguinte forma: (posições vazias) x (peças diferentes disponíveis) – (jogadas não possíveis por peças do adversário). No início do jogo, quando o tabuleiro está vazio, são possíveis 64 jogadas ($16 \times 4 - 0$).

Depois de encontradas as jogadas possíveis com o predicado **setof** (mostrado na figura 8), o computador faz **random_member** da lista de jogadas obtendo um elemento para jogar. Dado que o **setof** remove os duplicados, teoricamente, todas as jogadas possíveis têm a mesma probabilidade de serem escolhidas.

Níveis 2 e 3

```
14. valid_moves(Level, Board, White_Pieces, Brown_Pieces, Player, List_Of_Moves) :-  
15.   getPiecesAvailable( White_Pieces, Brown_Pieces, New_White_Pieces, New_Brown_Pieces),  
16.   setof([Value|Move], (valid_move(0, Move, Player, Board, New_White_Pieces, New_Brown_Pieces),  
17.     calc_value(Level, Board, Player, Move, New_White_Pieces, New_Brown_Pieces, Value)), Value_List_Of_Moves),  
18.   nth0(0, Value_List_Of_Moves, [Value | _One_Most_Value_Move]),  
19.   setof(Move1, member([Value | Move1], Value_List_Of_Moves), List_Of_Moves).
```

Figura 9 - Código para obter todas as jogadas possíveis de maior valor para os níveis 2 e 3 (Ficheiro: controller/computer.pl)

Quando o jogador escolhe níveis superiores a 1, surgem notórias diferenças no modo de jogar do computador. É feita uma avaliação para cada jogada possível de forma a escolher a melhor no momento. Esta avaliação, realizada pelo predicado **calc_value**, que é feita da seguinte forma:

1. Aplica-se a jogada num tabuleiro temporário;
2. Avalia-se esse tabuleiro usando o predicado **value** (descrito posteriormente na secção Avaliação do tabuleiro).

O código da figura 9 começa por remover (linha 14) as peças que se encontram repetidas e remover os 0s (que simbolizam peças vazias), de forma a melhorar o desempenho pois remove avaliações desnecessárias e repetidas. Na linha 15-17 o setof coloca no Value_List_of_Moves uma lista de membros do tipo [Value |Move], sendo Value a avaliação do Move.

Na linha 17 obtemos o melhor valor (o valor da primeira jogada desta Lista ordenada por melhor valor). Sendo que o setof ordena a lista por ordem crescente, foram usados valores menores para classificar as melhores jogadas.

Na linha 18, são seleccionadas da Value_List_of_value todos os Moves com valor igual ao melhor obtido da melhor jogada. Desta forma, apenas são retornados na List_Of_Moves os Moves que têm melhor valor.

Execução de jogadas

Uma jogada, antes de ser efetuada tem que ser verificada. A validação e execução de uma jogada num tabuleiro, obtendo o novo estado do jogo é feita com o predicado move.

```
150. move(Show_Error_Message, Move, Board, White_Pieces, Brown_Pieces, Player, New_Board, New_White_Pieces, New_Brown_Pieces) :-
151.     valid_move(Show_Error_Message, Move, Player, Board, White_Pieces, Brown_Pieces),
152.     move_piece(Move, Board, New_Board),
153.     remove_piece(Move, Player, White_Pieces, Brown_Pieces, New_White_Pieces, New_Brown_Pieces).
```

Figura 10 – Implementação do predicado move (Ficheiro: controller/game.pl)

Este predicado chama **valid_move** (explicada com mais detalhe mais à frente) para verificar a validade da jogada (linha 128), chama **move_piece** para realizar a jogada criando um novo tabuleiro (linha 129) e remove a peça entre as disponíveis (linha 130), com **remove_piece**. Esta remoção é feita trocando a representação numérica da peça por 0 e não por eliminação do elemento da lista. A remoção da peça da lista afetaria o predicado display_game que usa as listas New_White_Pieces e New_Brown_Pieces para mostrar as peças disponíveis, sendo necessário manter o tamanho destas listas e posição na lista das peças disponíveis (figura 7) de maneira a não desconfigurar o tabuleiro das peças disponíveis.

Validação de jogadas para jogadores

```
15. valid_move(1, [Row,Column,Piece], Player, Board, White_Pieces, Brown_Pieces) :-  
16.   is_cell_empty(Board, Row, Column, 1), !,  
17.   valid_play(Board, Row, Column, Piece, 1), !,  
18.   is_piece_available(Player, Piece, White_Pieces, Brown_Pieces, 1).
```

Figura 11 – Código utilizado para obter uma jogada possível para um computador (Ficheiro: controller/verifications.pl)

Para que uma jogada seja considerada válida, é necessário verificar se a posição e a peça inseridas pelo jogador são válidas, isto é, se tanto a linha como a coluna têm valores entre 1 e 4 e a peça é um cone, cylinder, sphere ou cube. De forma a ser capaz de lidar com inputs inválidos, a leitura dos inputs é sempre feita com **get_char** e **get_code** (conforme desejados chars ou números) seguido de **ship_line** para remover o resto do conteúdo do buffer (se existir). A validação das linhas e colunas é feita com o predicado **between** e a confirmação da validade das peças é feita pelo predicado **translate**, que simultaneamente confirma a validade da peça como a traduz num valor numérico de valor 11, 51, 71 e 91 no caso das peças brancas ou de 12, 52, 72 e 92 no caso das peças castanhas, se a peça for válida. Estas verificações não são realizadas no **valid_move** para jogadas de pessoas dado que já o são feitas logo após a jogada ser pedida ao utilizador, pois facilitam a correção do erro antes de mais inputs serem feitos.

Quando uma posição é válida, verifica-se posteriormente se a mesma não está a ser ocupada por outra peça (**is_cell_empty** - linha 16).

Posteriormente, confirmar-se que a regra do jogo não está a ser quebrada, isto é, que não existem peças do adversário, com a mesma forma que a peça que se pretende jogar, na mesma linha, coluna ou quadrado de extremidade (**valid_play** - linha 18). Por fim, na linha 20, verifica-se se a peça está disponível (**is_piece_available**), dado que cada jogador apenas possui duas peças de cada forma não podendo jogar as peças do adversário.

De forma a tornar o jogo mais user friendly, quando uma das condições não é cumprida é mostrada uma mensagem de erro adequada, dizendo qual o problema com a jogada do jogador.

Exemplo com da verificação **is_cell_empty**(Show_Error_Message, +Board,+Row, +Column):

```
25. is_cell_empty(Board, Row, Column, _Show_Error_Message) :-  
26.   get_piece_from_board(Row, Column, Board, Piece),  
27.   Piece == 0.  
28.  
29. is_cell_empty(_Board, Row, Column, 1) :-  
30.   not_empty_message(Row, Column), % Displays message to user  
31.   fail.
```

Figura 12 – Código que verifica se a célula não está ocupada (Ficheiro controller/verifications.pl)

O primeiro predicado é o que realiza a verificação, obtendo a peça do tabuleiro que ocupa a posição na qual se pretende fazer uma jogada e verifica se realmente está vazia, isto é, se a célula é 0. Caso esta condição não se verifique, o predicado falha, tentando o segundo predicado com o mesmo nome.

O segundo predicado tem como objetivo mostrar uma mensagem de erro dizendo ao utilizador que a posição introduzida já se encontra ocupada. Este predicado só será chamado se o primeiro falhar (porque a posição não está livre) e se o primeiro argumento for 1, que, de facto, indica que se quer mostrar a mensagem de erro. O argumento `Show_Error_Message` está a cabeça para ser o primeiro a falhar e evitar atribuições desnecessárias.

Dada a análise do predicado **is_cell_empty** e sabendo que os predicados **valid_play** e **is_piece_available** têm uma implementação semelhante, isto é, que segue o mesmo raciocínio do predicado analisado, conseguimos agora entender que os cuts (!) são então necessários para o correto funcionamento do programa sendo, por isso, cuts vermelhos. Esta necessidade provém do facto de, quando uma das condições falha, o programa não deve voltar a executar um predicado de verificação que já tenha sido executado e sucedido, evitando assim mensagens de erro inadequadas. De uma maneira simplificada, sem os cuts, se o programa estivesse a verificar uma jogada válida para uma célula vazia, porém a peça não estivesse disponível, aparecer-nos-ia no ecrã três mensagens de erro, em vez de apenas uma ('You don't have any more of those pieces. Choose a different one'). Isto aconteceria sem os cuts, dado que o processo de backtracking iria tentar encontrar outras soluções para o **valid_play** e seguidamente **is_cell_empty**, nos predicados que apenas têm por objetivo mostrar uma mensagem de erro e seguidamente falham.

Podemos encontrar cuts com a mesma função no predicado **get_move**.

Validação de jogadas para computador

```
9. valid_move(0, [Row,Column,Piece], Player, Board, White_Pieces, Brown_Pieces) :-  
10.  between(1, 4, Row), between(1, 4, Column), valid_piece(Piece),  
11.  is_cell_empty(Board, Row, Column, 0),  
12.  valid_play(Board, Row, Column, Piece, 0),  
13.  is_piece_available(Player, Piece, White_Pieces, Brown_Pieces, 0).
```

Figura 13 – Código usado pelo computador para obter jogadas válidas (Ficheiro: controller/verifications.pl)

Esta outra versão do predicado **valid_move** é necessária dado que, quando o computador executa este predicado, este pretende obter como solução uma jogada (em Move) que seja válida, o que leva há necessidade de algumas diferenças:

1. A linha 10 passa a ser necessária visto que **Move** não vem instanciado anteriormente, sendo então necessário dar valores válidos ao Move, para que, por backtracking, o programa seja capaz de avaliar todas as jogadas possíveis.
2. Quando o predicado falha, não se quer que sejam mostradas mensagens de erro, dado que é o computador que o está a executar.
3. Os cuts (!) precisam de ser removidos pois impedem o funcionamento desejado do setof, que procura todas as soluções por backtracking.

Final do jogo

```
60. game_over(_Show_Message, Board, _Winner, [Row|[Column|_Piece]], _White_Pieces,
    _Brown_Pieces, _Mode) :-
61.   get_row_sum(Board, Row, Row_Sum),
62.   Row_Sum =\= 22,
63.   get_column_sum(Board, Column, 0, Col_Sum),
64.   Col_Sum =\= 22,
65.   get_square_num(Row, Column, Square_Num),
66.   get_square_sum(Square_Num, Board, Square_Sum),
67.   Square_Sum =\= 22.
68.
69. game_over(1, Board, Winner, _Move, White_Pieces, Brown_Pieces, Mode) :-
70.   display_game(Board, 0, White_Pieces, Brown_Pieces),
71.   congratulate_winner(Winner, Mode),
72.   get_interaction,
73.   play.
```

Figura 14 – Código para verificar o fim do jogo (Ficheiro: controller/verifications.pl)

O primeiro **game_over** a ser feito é sucedido se o jogo ainda não tiver terminado. A verificação do fim do jogo é feita através da soma dos elementos. Dado que um jogador pode ganhar com peças do adversário, a soma não tem em conta o último número da peça, apenas o primeiro, que identifica a forma. Foram usados valores específicos para a representação interna de cada uma das peças de forma a que a sua soma seja um valor impossível de obter com qualquer outro conjunto de peças. Sendo que o cilindro é representado pelo valor 1, o cubo por 5, o cilindro por 7 e a esfera por 9, uma linha, coluna ou quadrado que possui as 4 formas diferentes tem um valor total de 22.

De forma a evitar verificações desnecessárias, é passada no argumento Move a última jogada realizada. Assim, evita-se fazer uma verificação de final do jogo no tabuleiro inteiro, sendo apenas necessário percorrer a linha, coluna e quadrado extremidade onde a jogada foi realizada. Se alguma das somas for igual a 22, o jogo termina.

Quando o jogo termina e o **game_over** falha no primeiro predicado, o segundo é executado, fazendo display do estado final do jogo e felicitando o vencedor indicando o mesmo e pontuação atual dos dois.

Avaliação do tabuleiro

Para serem realizadas melhores jogadas num nível de dificuldade de computador superior a 1, é usado um predicado de avaliação que avalia um estado de tabuleiro segundo a perspetiva do jogador atual.

Foram implementadas várias condições de avaliação do tabuleiro, sendo que o nível 3 difere do nível 2 na medida em que realiza um maior número de condições, o que, no geral, melhora a qualidade da jogada escolhida.

Os cuts (!) usados têm por objetivo impedir que o funcionamento de backtracking do **setof** execute as próximas avaliações quando a atual jogada já foi .

Avaliação 1

```
42. value(_Level, Board, _Player, Move, _White_Pieces, _Brown_Pieces, -66) :-  
43.   not(game_over(0, Board, _Player, Move, _White_Pieces, _Brown_Pieces, _Mode)),  
44.   !.
```

Figura 15 – Código para avaliar um tabuleiro (Ficheiro: controller/computer.pl)

A nossa primeira avaliação de um tabuleiro verifica se este se apresenta num estado vencedor. A(s) jogada(s) vitoriosas são avaliadas com o valor **-66**.

Avaliação 2

```
47. value(_Level, Board, Player, _Move, White_Pieces, Brown_Pieces, 10) :-  
48.   change_player(1, Player, New_Player),  
49.   setof(Move, (valid_move(0, Move, New_Player, Board, White_Pieces, Brown_Pieces),  
50.     move_piece(Move, Board, New_Board),  
51.     not(game_over(0, New_Board, _New_Player, Move, _White_Pieces, _Brown_Pieces, _Mode))), _List_Of_Moves),  
52.   !.
```

Figura 16 – Código para avaliar um tabuleiro (Ficheiro: controller/computer.pl)

A avaliação seguinte é um pouco mais defensiva, procurando ver se existem jogadas vitoriosas para o adversário. Estas jogadas são as piores que podemos efetuar sendo avaliadas com **10**, de forma a serem colocadas no fim da lista de jogadas possíveis para que sejam evitadas. Estas jogadas são guardadas pois podemos encontrarmo-nos numa situação em que todas as nossas jogadas acabam numa vitória do adversário.

Avaliação 3

```
55. value(_Level, Board, Player, _Move, White_Pieces, Brown_Pieces, -65) :-  
56.   change_player(1, Player, New_Player), % gets the number of the other Player  
57.   setof([Row, Column, Piece], (valid_move(0, [Row, Column, Piece], Player, Board, White_Pieces, Brown_Pieces),  
58.     get_opposite(Other_Piece, Piece),  
59.     not(valid_move(0, [Row, Column, Other_Piece], New_Player, Board, White_Pieces, Brown_Pieces))),  
60.     move_piece([Row, Column, Piece], Board, New_Board),  
61.     not(game_over(0, New_Board, _Player, [Row, Column, Piece], _White_Pieces, _Brown_Pieces, _Mode))), _List_Of_Moves),  
62.   !.
```

Figura 17 – Código para avaliar um tabuleiro (Ficheiro: controller/computer.pl)

Dadas as regras de jogo, é possível efetuar uma jogada que coloca três peças diferentes numa linha ou coluna ou quadrado exterioridade, permitindo-nos ganhar numa próxima jogada com uma jogada que não pode ser efetuada pelo adversário. Quando uma jogada deste tipo é encontrada, a vitória é garantida na nossa futura jogada. Estas jogadas são avaliadas com **-65**.

Avaliação 4

67. value(2, _Board, _Player, _Move, _White_Pieces, _Brown_Pieces, 0).

Figura 19 – Código para avaliar um tabuleiro no nível 2 (Ficheiro: controller/computer.pl)

Se nenhuma das verificações de avaliação anteriores se confirmar, todas as jogadas são avaliadas com 0 quando o computador está num nível de dificuldade de 2.

Avaliação 5

```
72. value(3, Board, Player, _Move, White_Pieces, Brown_Pieces, Value) :-  
73.   change_player(1, Player, New_Player),  
74.   not(setof(Move, (valid_move(0, Move, New_Player, Board, White_Pieces, Brown_Pieces),  
75.     move_piece(Move, Board, New_Board),  
76.     not(game_over(0, New_Board, _New_Player, Move, _White_Pieces, _Brown_Pieces, _Mode))), _List_Of_Moves)),  
77.   setof(Move, valid_move(0, Move, New_Player, Board, White_Pieces, Brown_Pieces),  
78.     List_Of_Moves),  
79.   length(List_Of_Moves, Length),  
79.   Value is -64 + Length.
```

Figura 19 – Código para avaliar um tabuleiro (Ficheiro: controller/computer.pl)

Enquanto que no nível 2, quando uma jogada não verifica nenhuma das condições anteriores, esta é avaliada com **0**, no nível 3 todas as jogadas são avaliadas segundo um critério. Esse critério é o número de jogadas (variável **Length** na linha 74) que restam ao jogador adversário com esta jogada. Este critério parece inocente numa primeira vista, no entanto, observando-o atentamente, implica 2 circunstâncias:

1. São jogadas apenas peças em linhas, colunas e quadrados que não as contém, aumentando as nossas chances de ganhar com as mesmas.
2. Aproximámo-nos de uma jogada pré-vencedora, por aumentar o impedimento de jogadas do adversário.

Jogada do computador

A escolha da jogada a efetuar pelo computador é feita através do predicado **choose_move**, que obtém todas as jogadas possíveis com **valid_moves** e depois usa o predicado **get_move**, para selecionar de forma aleatória uma dessas jogadas.

```
3. choose_move(Board, White_Pieces, Brown_Pieces, Level, Move, Player, _Show_Error_Message) :-  
4.   valid_moves(Level, Board, White_Pieces, Brown_Pieces, Player, List_Of_Moves),  
5.   get_move(Level, List_Of_Moves, Move).
```

Figura 20 – Código para escolher uma jogada para o computador (Ficheiro: controller/computer.pl)

Quanto maior for o valor de **Level**, melhores serão as jogadas efetuadas. Esta relação é possível devido ao predicado **valid_moves**, que, como mencionado anteriormente, se responsabiliza pela obtenção de jogadas consoante o nível de dificuldade.

Independentemente do nível, o predicado **get_move** apenas faz um **random_member**, obtendo-se uma jogada aleatória entre as avaliadas com maior valor. Por exemplo, se houver mais que uma jogada vencedora no tabuleiro o **random_member** escolhe uma jogada aleatória entre as jogadas vencedoras.

Conclusões

Quando iniciamos o projeto estávamos cientes que Prolog usa uma lógica e programação diferente das que tínhamos aprendido até agora, e que devíamos mudar a forma como pensamos para o sucesso do trabalho. À medida que o projeto foi avançando verificamos que esta linguagem tem vantagens em relação às aprendidas até agora para a programação de jogos como este, baseados em restrições. No fim do projeto procuramos melhorar a eficiência do programa.

O desenvolvimento do projeto foi conforme o planeado, esperando numa cadeia futura aprender a criar as árvores de jogadas possíveis características da inteligência artificial, de forma a obter um computador verdadeiramente inteligente e mais poderoso que o do nosso programa.

Bibliografia

9. 1jour-1jeu.com. (2019). *Quantik (2019) - Board games - 1jour-1jeu.com*. [online] Available at: <https://en.1jour-1jeu.com/boardgame/2019-quantik/> [Accessed 1 Oct. 2019].
10. BoardGameGeek. (2019). *Quantik*. [online] Available at: <https://boardgamegeek.com/boardgame/286295/quantik/credits> [Accessed 6 Oct. 2019].
11. Mastersofgames.com. (2019). *Quantik by Gigamic | Abstract Strategy Game for 2 Players*. [online] Available at: <https://www.mastersofgames.com/cat/board/gigamic-quantik-game.htm> [Accessed 7 Oct. 2019].
12. YouTube. (2019). *GIGAMIC - QUANTIK*. [online] Available at: <https://www.youtube.com/watch?v=Ft-4FJpZG7Q&> [Accessed 9 Oct. 2019].