U.PORTO
FEUP FACULDADE DE ENGENHARIA
UNIVERSIDADE DO PORTO

Programação em lógica

Relatório - CAM

3MIEIC06-grupo 1

2018/2019

Trabalho realizado por:

- Maria Helena Sampaio de Mendonça Montenegro e Almeida up201604184@fe.up.pt
- Juliana Maria Cruz Marques up201605568@fe.up.pt

Índice

1. Introdução

Este projeto foi desenvolvido no âmbito da unidade curricular de Programação em lógica no sistema de desenvolvimento SICtus Prolog, com o objetivo de implementar, nesta linguagem um jogo de tabuleiro, no nosso caso: CAM.

Este relatório encontra-se dividido em secções:

- **O Jogo CAM:** Descrição do jogo, história e as suas regras.
- **Lógica do Jogo:**
  - **Representação do Estado do Jogo:** Exemplificação de estados iniciais, intermédios e finais do jogo.
  - **Visualização do Tabuleiro:** Descrição do predicado de visualização.
  - **Lista de Jogadas Válidas**: Descrição dos predicados usadas para a validação das jogadas.
  - **Execução de Jogadas**: Explicitação do ciclo do jogo e de como é executada cada jogada.
  - **Avaliação do Tabuleiro:** Descrição dos predicados que avaliam o conteúdo do tabuleiro.
  - **Final do Jogo:** Descrição dos predicados que verificam o fim de jogo.
  - **Jogada do Computador:** Descrição dos predicados de geração de movimentos inteligentes por parte do computador.
- **Interface com o Utilizador.**

2. O jogo – CAM

2.1. História

Em 1888, foi publicado o jogo de estratégia Chivalry, por George S. Parker, fundador da companhia Parker Brothers. Este jogo foi baseado no xadrez, sendo mais fácil de jogar, por envolver menos tipos de peças e movimentos.

Este jogo não teve o êxito esperado, pelo que Parker continuou a desenvolvê-lo e a reduzir o seu tamanho, originando uma versão simplificada, que se num dos jogos mais populares lançados pela companhia: **Camelot**, 1930. No entanto, as simplificações não ficaram por aí, tendo sido desenvolvidas bastantes variações do jogo. Entre elas, a mais popular: **Cam**.

Cam é um jogo de tabuleiro de estratégia de dois jogadores, desenvolvido por Leroy Howard, em 1949. Demora cerca de 30 minutos a ser jogado, e tem apenas dois tipos de peças, cavaleiros e soldados.

2.2. Tabuleiro

O jogo é jogado num tabuleiro com uma disposição especial com 67 células. Existem dois jogadores, o jogador branco e o jogador preto.

Existem duas células chamadas castelos (castelo branco – célula D1 e castelo preto – célula D13). Cada jogador no início do jogo possui 7 peças: 5 homens e 2 cavaleiros.

Fig 1. Tabuleiro de CAM

2.3. Objetivo

Existem duas maneiras de vencer o jogo:

- O jogador vence se capturar todas as peças do adversário.
- O jogador vence se conseguir mover uma das suas peças (cavaleiro ou homem) para o castelo do adversário.

2.4. Regras

O jogo começa sempre com um movimento do jogador branco, a partir daí as jogadas ocorrem alternadamente entre jogadores.

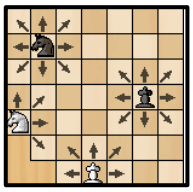


Fig. Exemplo de movimentos simples

Possíveis movimentos feitos por cavaleiros e homens:

- ***Movimento simples:***

A peça pode-se deslocar horizontalmente, diagonalmente ou verticalmente para uma célula adjacente não ocupada por outra peça.

- ***Galope:***

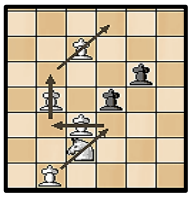


Fig 2. Exemplo de um galope

A peça pode saltar horizontalmente, diagonalmente ou verticalmente sobre qualquer outra peça do mesmo jogador desde que haja uma célula vazia logo após.

As peças galopadas não são removidas do tabuleiro. Pode ser feito mais do que um salto sobre as peças desde que possível podendo o movimento ter direção variada, não sendo obrigatório. É proibido fazer galope circular, ou seja, começar numa célula e terminar na mesma célula inicial.

- ***Captura:***

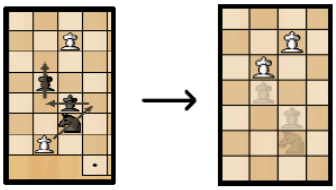


Fig 3. Exemplo de uma captura

Uma peça pode saltar sobre uma qualquer peça do oponente posicionada em uma célula horizontal, vertical ou diagonalmente adjacente, se houver uma célula vazia na mesma direção logo depois desta. As peças do adversário saltadas são capturadas e removidas do tabuleiro.

Se uma peça pode continuar a saltar sobre as peças do adversário é obrigada a fazê-lo, sendo permitido neste caso o salto circular, ou seja, a peça jogada pode acabar na mesma posição em que começou.

A captura é obrigatória. A única situação em que um jogador pode ignorar a sua obrigação de capturar é quando, em seu movimento anterior, ele capturou uma das peças do adversário no seu castelo, terminando a sua vez lá, no próximo turno, é obrigado a retirar a peça do seu castelo.

Movimento apenas possível com cavaleiro:

- ***Ataque do cavaleiro:***

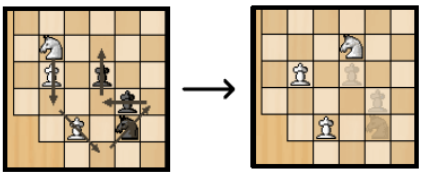


Fig 4. Exemplo de um ataque do cavaleiro

É a combinação de captura e galope num único movimento, quando um cavaleiro faz este movimento é obrigatório ser feito primeiro um galope e só depois a captura.

Este movimento não é obrigatório, a única situação que obriga este movimento é quando um cavaleiro faz galope ao lado de uma peça do adversário que pode ser capturada, quando isto acontece o cavaleiro é obrigado a capturar a peça do adversário a não ser que o cavaleiro possa continuar a fazer galope e capture mais do que uma peça do adversário.

Notas:

- É proibido mover uma peça para o próprio Castelo através de um movimento simples ou de um galope, mas um jogador pode mover uma das suas peças para o seu próprio Castelo saltando sobre a peça do adversário ao lado do Castelo.
- É proibido mover um Cavaleiro para o Castelo do próprio jogador durante a parte do galope de um Cavaleiro. Um jogador que terminar seu turno pulando em seu próprio Castelo deve imediatamente mover aquela peça do seu Castelo no próximo turno (seja por captura ou por movimento simples ou galope).

3. Lógica de jogo

3.1. Organização do código

O código está organizado nos seguintes ficheiros:

- **cam.pl** - ficheiro principal que consulta todos os ficheiros e as bibliotecas necessárias. Contém a função play, que inicia o jogo.
- **menu.pl** - ficheiro que contém todas as funções que permitem mostrar o menu, tal como as funções que analizam o input do jogador relativo a cada uma das opções do menu.
- **display.pl** - ficheiro que contém as funções responsáveis por mostrar o tabuleiro e a vez do jogador.
- **input.pl** - ficheiro responsável por receber a célula a partir da qual o jogador humano se quer mover, verificando se existe nessa célula uma peça desse mesmo jogador, e a célula para onde se quer mover.
- **useful_functions.pl** - ficheiro que contém algumas funções auxiliares importantes, tal como funções para alterar o tabuleiro e verificar que símbolo se localiza numa determinada célula do tabuleiro, e alguns dados necessários, que incluem as células onde se localizam os castelos de cada jogador, quais os símbolos de cada jogador, quais os cavaleiros de cada jogador e quem são os jogadores.
- **game_logic.pl** - ficheiro que contém o ciclo do jogo, contendo as funções que permitem a vez de cada jogador, as funções move, valid_moves e game_over, e outras funções auxiliares necessárias.
- **check_symbol.pl** - ficheiro que contém a função check_symbol que auxilia a função valid_moves. Esta função verifica se uma célula que poderá ser destino de um movimento está livre para o tal movimento e verifica se este movimento é adjacente, é um salto sobre uma peça inimiga ou é um salto sobre uma peça amiga.
- **bot.pl** - ficheiro que contém as funções necessárias ao movimento do computador, incluindo a função choose_move e a função value.

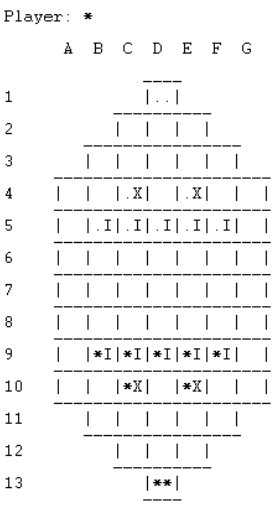
3.2. Representação do estado interno de jogo

O tabuleiro de jogo tem 67 células, e 16 peças (7 peças para cada jogador), 2 cavaleiros brancos, 2 cavaleiros pretos, 5 homens brancos e 5 homens pretos. Decidimos representar as diferentes peças e células pelos seguintes símbolos:

- Espaço exterior ao tabuleiro – internamente (‘n’) visualização (‘ ’);
- Célula vazia – internamente (0), visualização (| |);
- Cavaleiro branco – internamente (1), visualização (.X);
- Cavaleiro preto – internamente (4), visualização (*X);
- Homem branco – internamente (2), visualização (.I);
- Homem preto – internamente (3), visualização (*I);
- Castelo branco – internamente (5), visualização (..);
- Castelo preto – internamente (6), visualização (**);

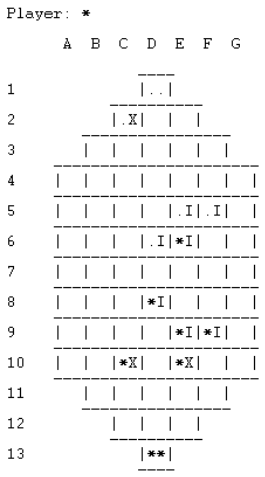
```
players_pieces('.', 1).
players_pieces('.', 2).
players_pieces('*', 3).
players_pieces('*', 4).

knight('.', 1).
knight('*', 4).

enemy('.', '*').
enemy('*', '.').

castle('.','D'-1).
castle('*','D'-13).
```

3.2.1. Representação do estado inicial do tabuleiro



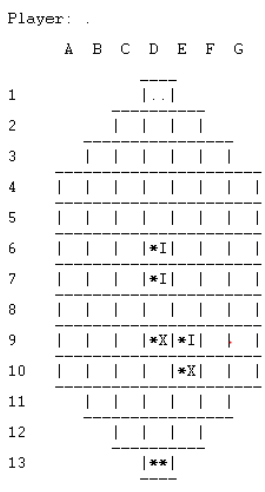

```
initial_board([[n,n,n,n,5,n,n,n,n],
       [n,n,n,0,0,0,n,n,n],
       [n,n,0,0,0,0,0,n,n],
       [n,0,0,1,0,1,0,0,n],
       [n,0,2,2,2,2,2,0,n],
       [n,0,0,0,0,0,0,0,n],
       [n,0,0,0,0,0,0,0,n],
       [n,0,0,0,0,0,0,0,n],
       [n,0,3,3,3,3,3,0,n],
       [n,0,0,4,0,4,0,0,n],
       [n,n,0,0,0,0,0,n,n],
       [n,n,n,0,0,0,n,n,n],
       [n,n,n,n,6,n,n,n,n]]).
```

3.2.2. Representação de um possível estado intermédio




```
intermediate_board([[n,n,n,n,5,n,n,n,n],
      [n,n,n,1,0,0,n,n,n],
      [n,n,0,0,0,0,0,n,n],
      [n,0,0,0,0,0,0,0,n],
      [n,0,0,0,0,2,2,0,n],
      [n,0,0,0,2,3,0,0,n],
      [n,0,0,0,0,0,0,0,n],
      [n,0,0,0,3,0,0,0,n],
      [n,0,0,0,0,3,3,0,n],
      [n,0,0,4,0,4,0,0,n],
      [n,n,0,0,0,0,0,n,n],
      [n,n,n,0,0,0,n,n,n],
      [n,n,n,n,6,n,n,n,n]]).
```

3.2.3. Representação de um possível estado final




```
final_board([[n,n,n,n,5,n,n,n,n],
       [n,n,n,0,0,0,n,n,n],
       [n,n,0,0,0,0,0,n,n],
       [n,0,0,0,0,0,0,0,n],
       [n,0,0,0,0,0,0,0,n],
       [n,0,0,0,3,0,0,0,n],
       [n,0,0,0,3,0,0,0,n],
       [n,0,0,0,0,0,0,0,n],
       [n,0,0,0,4,3,0,0,n],
       [n,0,0,0,0,4,0,0,n],
       [n,n,0,0,0,0,0,n,n],
       [n,n,n,0,0,0,n,n,n],
       [n,n,n,n,6,n,n,n,n]]).
```

3.3. Visualização do tabuleiro

Segue-se o código desenvolvido para visualizar o tabuleiro:

A função ***traduz***, traduz os números representados internamente no tabuleiro para símbolos fáceis de diferenciar na interface com o utilizador:

A função ***display_game*** trata de imprimir todo o jogo, primeiramente será imprimido apenas o board inicial chamando esta função apenas com o *player*, depois recursivamente chamar-se-á recursivamente já com um *board*. Esta função tratará de escrever o jogador que vai jogar a seguir, depois chamará a função ***write_letters*** que irá imprimir as letras acima do tabuleiro através do seu código ASCII.

Seguidamente chama-se a função ***print_tab*** que imprime o tabuleiro. Esta chama as funções para imprimir os separadores entre linhas ***print_sep***, a função que imprime os números á esquerda do tabuleiro,numeram as linhas, ***print_num***, e ainda a função ***print_line*** que imprime recursivamente as linhas do tabuleiro, chamando a cada iteração a função ***print_cell*** que trata de traduzir e escrever cada célula do tabuleiro.

A função ***print_tab*** recebe o *board* que está a ser desenhado e um *Aux* representando o índice da linha que está sendo desenhada em cada iteração.

```
traduz(n, '   ').
traduz(0, '  |').
traduz(1, '.X|').
traduz(2, '.I|').
traduz(3, '*I|').
traduz(4, '*X|').
traduz(5, '..|').
traduz(6, '**|').
```

```
display_game(Player) :-
initial_board(B),
display_game(B, Player).
```

```
display_game(Board, Player) :-
sleep(1), nl, nl,
write('Player: '),
write(Player), nl, nl,
write('      '),
write_letters('A'), nl,
print_tab(Board, 0).
```

```
print_tab([], _Aux) :- nl,

write('              ----').
print_tab([L|T], Aux) :-
nl, write(' '),
print_sep(L, Aux), nl,
print_num(Aux),
print_line(L),
N is Aux+1,
print_tab(T, N).
```

3.4. Lista de jogadas válidas

Para verificar quais as jogadas possíveis dadas uma posição do tabuleiro, utilizamos a função ***valid_moves***. Esta função recebe o tabuleiro e o jogador atual e retorna uma lista com as jogadas válidas.

Esta função tem duas instâncias, uma que verifica se o jogador tem uma peça no seu próprio castelo, devolvendo como movimentos possíveis movimentos que retirem a peça do mesmo, e outra que calcula as jogadas possíveis correspondentes a movimentos para casas adjacentes, movimentos de salto sobre peças inimigas e movimentos de salto sobre peças do mesmo jogador.

A função ***check_board*** recebe uma letra e um número, posição da peça no tabuleiro, e retorna o símbolo presente nessa posição no tabuleiro. A função ***check_pawns*** verifica para cada peça do tabuleiro selecionada pelo jogador, as células adjacentes e guarda os movimentos válidos em 3 categorias:

- **ListOfMoves** - movimentos para peças adjacentes;
- **ListOfEnemies** – movimentos com saltos sobre os inimigos;
- **ListOfFriends** - movimentos com saltos sobre as peças do mesmo jogador (amigos).

A função ***check_pawns*** por sua vez chama a função ***ckeck_adjacents*** que percorre a lista de células adjacentes de uma peça localizada em *Letter-Number* e coloca em *Movimentos* os possíveis movimentos que podem ser feitos diretamente, para peças adjacentes, sobre inimigos (captura) e sobre peças do mesmo jogador.

```
valid_moves(Board, Player, ListOfMoves) :-
    castle(Player, L-N),
    check_board(Board, Symbol, L-N), players_pieces(Player, Symbol), !,
    check_pawns(Board, Player, [L-N], ListOfMovements, [], ListOfEnemies, [], ListOfFriends, []),
    return_list_of_moves(ListOfMoves, ListOfMovements, ListOfEnemies, ListOfFriends).

valid_moves(Board, Player, ListOfMoves) :-
    get_pawns(Board, Player, ListOfPawns),
    check_pawns(Board, Player, ListOfPawns, ListOfMovements, [], ListOfEnemies, [], ListOfFriends, []), !,
    return_list_of_moves(ListOfMoves, ListOfMovements, ListOfEnemies, ListOfFriends).

check_pawns(_, _, [], ListOfMoves, Aux1, ListOfEnemies, Aux2, ListOfFriends, Aux3) :-
    ListOfMoves = Aux1,
    ListOfEnemies = Aux2,
    ListOfFriends = Aux3.

check_pawns(Board, Player, [H|T], ListOfMoves, Aux1, ListOfEnemies, Aux2, ListOfFriends, Aux3) :-
    adjacent_cells(H, List),
    check_adjacents(Board, Player, H, List, Moves, Aux1, Enemies, Aux2, Friends, Aux3),
    check_pawns(Board, Player, T, ListOfMoves, Moves, ListOfEnemies, Enemies, ListOfFriends, Friends).

check_board(Board, Symbol, Letter-Number) :-
        Number =< 13, char_code(Letter, Num),
        get_char_code_A_G(NumA, NumG),
        Num >= NumA, Num=<NumG,
        N is Num-NumA+1, N2 is Number-1,
        access_board(Board, Symbol, N, N2).
```

3.5. Execução de jogadas

O predicado responsável pela execução de jogadas é o ***move***, este predicado move a peça localizada em *Letter-Number* para a posição *Newletter-NewNumber* se possível, alterando assim o tabuleiro. Esta recebe a posição atual da peça, *Letter-Number*, a célula para o qual queremos mover a peça, *Newletter-NewNumber*, o tipo de jogador, *TypeOfPlayer*, pode ser jogador ou computador e recebe ainda o jogador que está a jogar, o tabuleiro atual e retorna o novo tabuleiro já alterado. É responsável também por verificar se o jogador, após jogar, pode jogar outra vez.

Para isto o predicado cham 6 predicados, por esta ordem:

- ***valid_moves*** - que vai verificar quias as jogadas possíveis dadas uma posição e peça, como explicado acima.
- ***member*** - retorna *yes*, se a nova posição estiver na lista de jogadas válidas, *ListOfMoves*.
- ***check_board*** - recebe a posição da peça no tabuleiro, e retorna o símbolo presente nessa mesma posição no tabuleiro.
- ***change_board*** – recebe o tabuleiro e altera o símbolo presente na célula de posição *Letter-Number*, para *Symbol*, retornando o tabuleiro alterado. Ora este predicado vai ser chamado 2 vezes, a primeira para colocar a posição da peça que queremos mover a 0, vazia, e a segunda chamada para colocar a peça na nova posição escolhida pelo jogador.

- ***display*** – verifica se o jogador pode ou tem de jogar novamente, mostrando o tabuleiro em caso positivo e chamando a função que permite uma nova jogada do mesmo jogador. A função ***display*** realiza as seguintes hipóteses:
    - Se a peça que o jogador usou na sua jogada saltou sobre uma peça do jogador adversário, e se essa peça ainda tem mais peças inimigas sobre as quais pode saltar, repete-se o turno do jogador. (Esta jogada diz respeito à Captura.)
    - Se o jogador for humano e a peça utilizada na sua jogada foi um cavaleiro que tenha saltado sobre uma peça do adversário e que tenha outras peças do mesmo jogador sobre as quais pode saltar, pergunta-se ao jogador se quer saltar ou passar a vez. (Esta jogada diz respeito ao Ataque do Cavaleiro).
    - Se a peça que o jogador utilizou na sua jogada for um cavaleiro que tenha saltado sobre uma peça do mesmo jogador e existem peças do adversário adjacentes sobre as quais o cavaleiro possa saltar, este salto é efetuado ao repetir-se a vez do jogador. (Esta jogada diz respeito ao Ataque do Cavaleiro.)
    - Se o jogador for humano, a peça utilizada saltou sobre uma peça do mesmo jogador e que essa peça ainda tenha outras peças do mesmo jogador adjacentes sobre as quais possam saltar, é perguntado ao jogador se deseja saltar ou passar a vez. (Esta jogada diz respeito ao Galope.)
    - Se o jogador for humano, a peça utilizada saltou sobre uma peça do mesmo jogador e que essa peça ainda tenha outras peças do mesmo jogador adjacentes sobre as quais possam saltar, o salto é realizado. (Esta jogada diz respeito ao Galope.)
    - Para o computador inteligente, não é utilizada a função ***display*** para verificar se a vez do jogador é repetida, mas a função ***check_if_same_turn***, localizada no ficheiro ***bot.pl***.

```
move(Letter-Number-NewLetter-NewNumber, TypeOfPlayer, Player, Board, NewBoard) :-
    valid_moves(Board, Player, ListOfMoves),
    member(Letter-Number-NewLetter-NewNumber, ListOfMoves),
    check_board(Board, Symbol, Letter-Number),
    change_board(Board, 0, Letter-Number, Board1),
    change_board(Board1, Symbol, NewLetter-NewNumber, Board2),
    display(TypeOfPlayer, Player, Letter-Number-NewLetter-NewNumber, Board2, NewBoard).

move(_, TypeOfPlayer, Player, Board, NewBoard):- write('Can\'t move. Try again.'),
    players_turn(Board, TypeOfPlayer, Player, NewBoard).
```

3.6. Final do jogo

Após cada jogada é fundamental verificar o estado do jogo, pois, a qualquer momento, um dos jogadores pode ganhar, isto é, caso o jogador esteja na casa do castelo inimigo, ou o inimigo não tem mais peças em jogo.

```
game_over(Board, Player) :- enemy(Player, Opponent),
    castle(Opponent, Letter-Number),
    check_board(Board, Symbol, Letter-Number),
    players_pieces(Player, Symbol),
    nl, nl, write('****GAME OVER****'), nl, write('**** Player '), write(Player), write(' wins****').

game_over(Board, Player) :- enemy(Player, Opponent),
    get_pawns(Board, Opponent, ListOfPawns),
    length(ListOfPawns, 0),
    nl, nl, write('****GAME OVER****'), nl, write('**** Player '), write(Player), write(' wins****').
```

3.7. Avaliação do tabuleiro

Para avaliarmos o tabuleiro, definimos o predicado ***value*** que recebe o tabuleiro, *Board*, o jogador, *Player* e retorna um valor associado a esse tabuleiro. Para o calculo do valor de cada tabuleiro, calculamos a distância entre o castelo adversário e a peça mais próxima, atribuindo-lhe um valor de 0 a 100, tendo este resultado um peso de 0.5. Calculamos a distância média entre as peças do jogador e o castelo adversário, atribuindo novamente um valor de 0 a 100 e um peso de 0.5. Para um resultado mais preciso, sempre que o tabuleiro tiver uma peça com um inimigo adjacente, são retirados 50 valores à pontuação.

```
value(Board, Player, Value) :-
    get_pawns(Board, Player, ListOfPawns),
    distance_pawns(Player, BestPawn, Aux2, 12, BestDistance, ListOfPawns),
    distance_pawns2(Player, ListOfPawns, AverageDistance, 0, 0),
    check_enemy_adjacent_recursive(Player, Board, ListOfPawns, V),
    Value is (12-AverageDistance)*8.3*0.5 + (12-BestDistance)*8.3*0.5+V.
```

3.8. Jogada do computador

Foram implementados dois níveis de dificuldade: um em que a jogada é escolhida aleatoriamente e outro onde são avaliados todos os tabuleiros possíveis para a próxima jogada e é escolhido o tabuleiro com a melhor avaliação.

```
choose_move(Difficulty, Player, Move, Board) :-
    Difficulty = 1,
    valid_moves(Board, Player, ListOfMoves),
    length(ListOfMoves, Len),
    random(0, Len, Int),
    nth0(Int, ListOfMoves, Move), !.

choose_move(Difficulty, Player, Board, NewBoard) :-
    Difficulty = 2,
    valid_moves(Board, Player, ListOfMoves),
    generate_boards(Board, Player, ListOfMoves, ListOfBoards),
    check_value(ListOfBoards, Board2, AuxBoard, BestValue, -100, Counter, 0, 0),
    nth0(Counter, ListOfMoves, Move),
    check_enemies_c2(Player, Move, Board, Board2, NewBoard).
```

4. Interface com o utilizador

A interface da linha de comandos foi feita de forma a proporcionar uma experiência agradável e simples ao utilizador.

O menu principal do jogo permite ao utilizador:

- Jogar, podendo escolher um de quatro modos de jogo:
  - **Player vs Player,** em que o jogador joga contra outro jogador, tendo controlo total do jogo.
  - **Pc vs Player**, em que o jogador jogará contra um bot, que poderá ser de dificuldade fácil (beginner) ou difícil(professional), sendo o pc o primeiro a jogar.
  - **Player vs Pc**, em que o jogador jogará contra um bot, que poderá ser de dificuldade fácil (beginner) ou difícil(professional), sendo o jogador o primeiro a jogar.
  - **Pc vs Pc**, em que o jogador poderá visualizar dois bots de dificuldades escolhidas previamente pelo utilizador, fácil (beginner) ou difícil (professional), a jogar.
- Aceder ás regras do jogo.
- Sair.

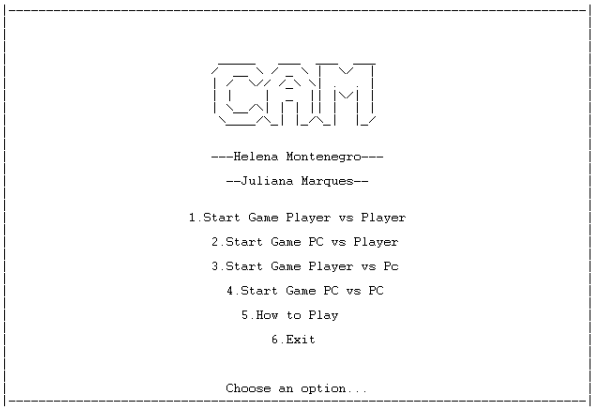


Fig 5. Menu principal

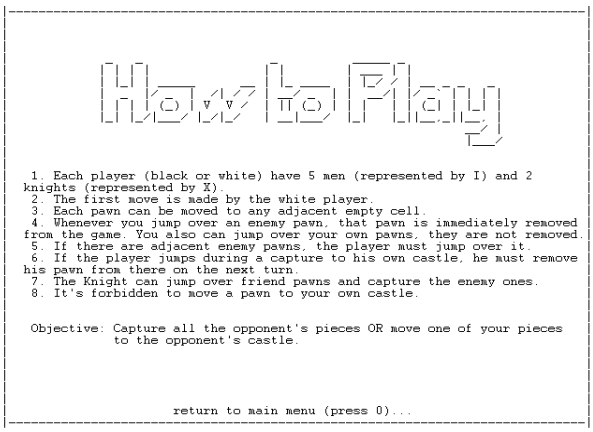


Fig 6. Página das regras de jogo

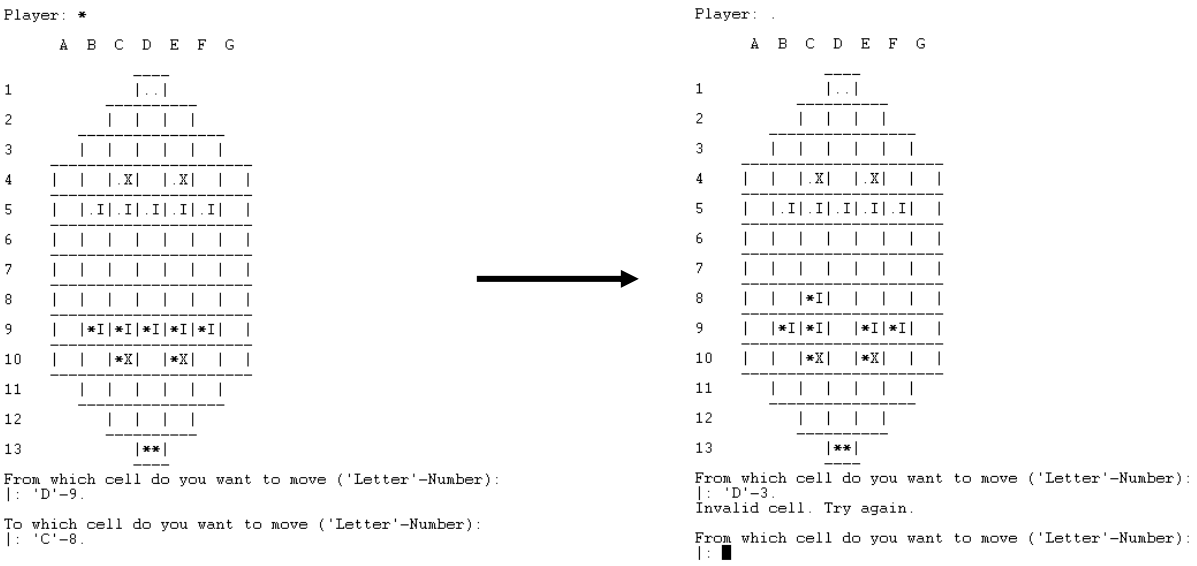


Fig 7. Exemplo da interface com o utilizador durante o jogo

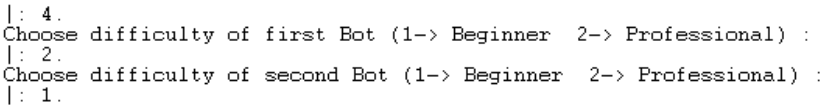


Fig 8. Quando selecionada a opção 2,3 e 4 do menu, é perguntado o nível de inteligência pretendido para o jogador de inteligência artificial, como mostra o exemplo acima.

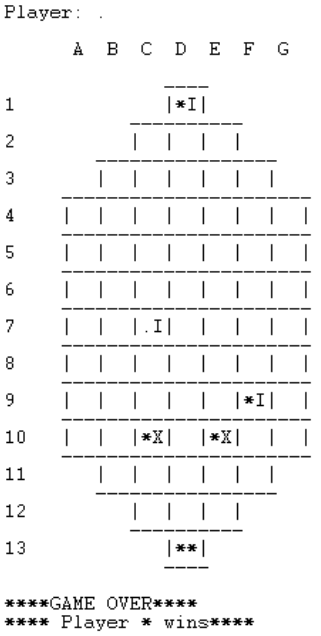


Fig 9. Exemplo da vitória do jogador *

5. Conclusões

O projeto teve como principal objetivo aplicar o conhecimento adquirido nas aulas teóricas e práticas, assim como o desenvolvimento de uma maneira de pensar diferente relativa a novo paradigma de programação, programação em lógica, tendo sido implementado um jogo de tabuleiro.

Ao longo do desenvolvimento deste projeto, foram encontradas algumas dificuldades, nomeadamente no pensamento recursivo e na escolha do melhor caminho a tomar em cada predicado.

Todos os pontos pedidos foram implementados com sucesso.

Em suma, o trabalho foi concluído com sucesso, e o seu desenvolvimento contribuiu positivamente para uma melhor compreensão da linguagem *Prolog*.

6. Instruções de compilação

Para proceder à execução do jogo, terá de ser consultado o ficheiro cam.pl. De seguida, basta inserir o comando *play*.

7. Bibliografia

- http://www.worldcamelotfederation.com/Cam.htm
- http://www.worldcamelotfederation.com/cam_prototype.htm
- http://thebiggamehunter.com/games-one-by-one/1815-2/
- https://www.thegamecrafter.com/games/cam
- http://www.swi-prolog.org/pldoc/doc/_CWD_/index.html