

Título do Trabalho

Relatório Final



Mestrado Integrado em Engenharia Informática e
Computação

Programação em Lógica

Grupo 04:

João Nuno Fonseca Seixas - 201505648
Renato Alexandre Sousa Campos - 201504942

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

12 de Novembro de 2017

Resumo

Resumo sucinto do trabalho com 150 a 250 palavras (problema abordado, objetivo, como foi o problema resolvido/abordado, principais resultados e conclusões).

Conteúdo

1	Introdução	4
2	O Jogo Campo Bello	4
3	Lógica do Jogo	6
3.1	Representação do Estado do Jogo	6
3.2	Visualização do Tabuleiro	6
3.3	Lista de Jogadas Válidas	8
3.4	Execução de Jogadas	8
3.5	Avaliação do Tabuleiro	9
3.6	Final do Jogo	9
3.7	Jogada do Computador	9
4	Interface com o Utilizador	9
5	Conclusões	9
	Bibliografia	10
A	Nome do Anexo	10

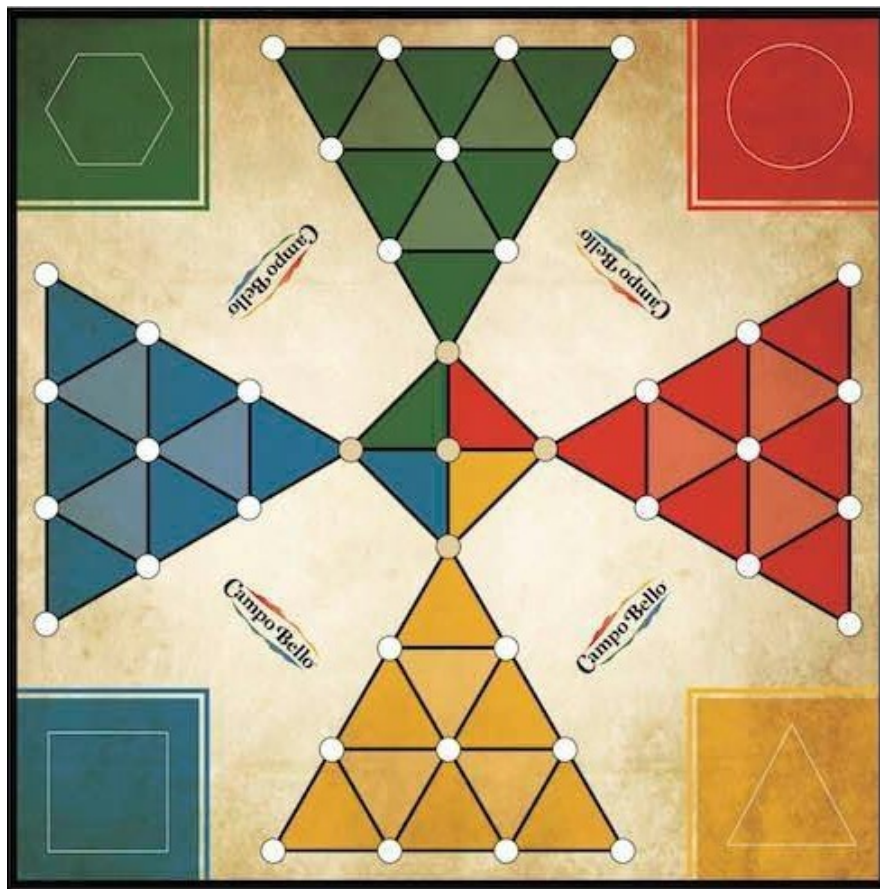
1 Introdução

O objetivo deste trabalho é implementar, em linguagem Prolog, um jogo de tabuleiro para dois jogadores e uma aplicação, em Prolog, para o jogar. O jogo permite jogar em três modos de utilização: Humano/Humano, Humano/Computador e Computador/Computador, e tem dois níveis de jogo para o Computador.

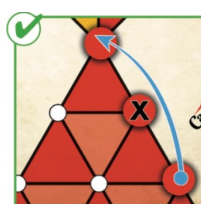
O relatório segue a seguinte estrutura: O Jogo Campo Bello, onde está descrito a história e principais regras de jogo; Lógica do Jogo, descrição do projeto e a implementação da lógica do jogo em Prolog; Interface com Utilizador, como é feita a interação Utilizador-Programa; Conclusões, principais conclusões e possíveis melhorias do trabalho desenvolvido.

2 O Jogo Campo Bello

Campo Bello é um jogo que pode ser jogado de 2 a 4 jogadores. Tem inspiração no jogo clássico "Resta Um" ou "Peg Solitaire" em Inglês. É um jogo ainda recente, criado em 2017. Para ganhar, um jogador deve tentar remover todas as suas peças do tabuleiro antes dos adversários. O tabuleiro consiste em 4 triângulos que rodeiam um diamante central. Os triângulos correspondem às áreas iniciais de cada jogador. As peças são removidas ao saltar: saltar uma peça nossa causa a remoção da peça que foi saltada; saltar uma peça adversária permite-nos remover uma peça nossa do tabuleiro. Na variante de apenas 2 jogadores, que vai ser implementada, os jogadores ficam com triângulos opostos e jogam alternadamente. É também possível executar saltos duplos e triplos numa só jogada. No fim do jogo cada jogador pontua 1 ponto por cada uma das suas peças fora da área inicial e 3 pontos por cada uma das suas peças dentro da sua área inicial. O jogador com menos pontos ganha!



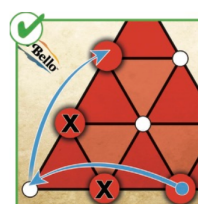
LEGAL MOVES



single jump



single jump

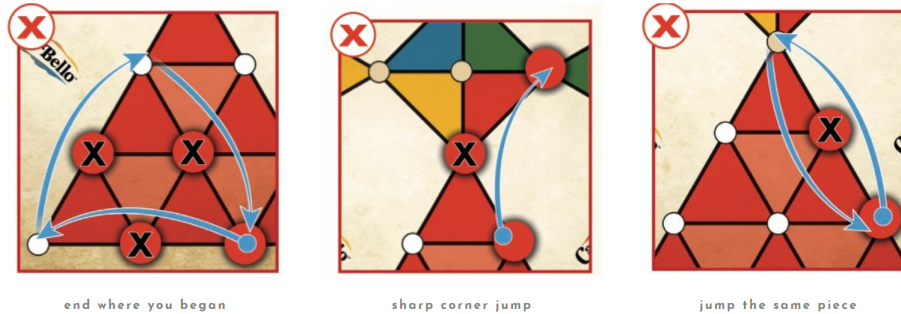


double jump



triple jump

ILLEGAL MOVES



<http://www.campobellogame.com/>

3 Lógica do Jogo

3.1 Representação do Estado do Jogo

O estado do jogo é representado por 2 listas que identificam em que posições do tabuleiro estão as peças de cada jogador (1 lista para cada jogador). As peças de cada jogador são designadas em Inglês por "movers".

Tabuleiro no estado de jogo inicial:

```
YMovers=[y1,y2,y3,y4,y5,y6,y7,y8,y9,g1,g2,g3,g4,g5,g6,g7,g8,g9]
BMovers=[r1,r2,r3,r4,r5,r6,r7,r8,r9,b1,b2,b3,b4,b5,b6,b7,b8,b9]
```

Tabuleiro no estado de jogo intermédio:

```
YMovers=[g6,g2,y4,y5,y6,y7,y8,y9]
BMovers=[y0,b8,r1,r3,r4,r5,r6,r7,r8,r9]
```

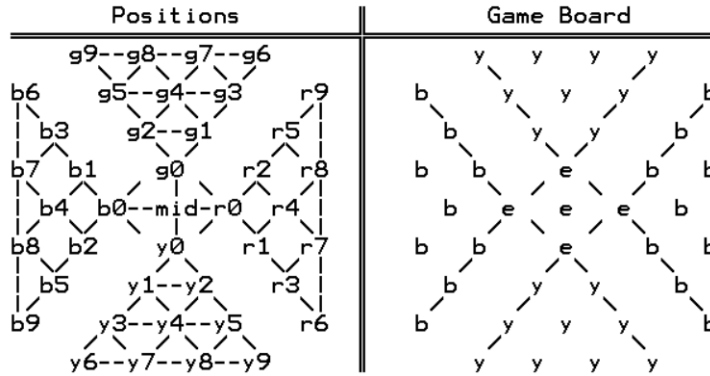
Tabuleiro no estado de jogo final:

```
YMovers=[g5,y5,y9,g6,g8,y6]
BMovers=[r5,r6,r8,y0,b8,b2,b3]
```

Com o decorrer do jogo as listas perdem elementos. O estado final do jogo é alcançado quando uma das listas estiver vazia ou quando não houver mais jogadas possíveis para o jogador que vai jogar.

3.2 Visualização do Tabuleiro

O predicado para visualização do tabuleiro mostra as posições do tabuleiro do lado esquerdo a usar para os movimentos e do lado direito o estado atual das peças. Usa *put_code* para melhorar o aspeto e facilitar a visualização.



```
displayBoard(Y,B) :-
    nl,
    write('          Positions          '), put_code(186), write('          Game Board'), nl,
    put_code(205),put_code(205),put_code(205),put_code(205),put_code(205),put_code(205),
    put_code(205),put_code(205),put_code(205),put_code(205),put_code(205),put_code(205),
    put_code(205),put_code(205),put_code(205),put_code(205),put_code(205),put_code(205),
    put_code(205),put_code(205),put_code(205),put_code(205),put_code(205),put_code(205),
    put_code(205),put_code(205),put_code(205),put_code(205),put_code(205),put_code(205),
    put_code(205),put_code(205),put_code(205),put_code(205),put_code(205),put_code(205),
    displayLine(1,Y,B),
    displayLine(2,Y,B),
    displayLine(3,Y,B),
    displayLine(4,Y,B),
    displayLine(5,Y,B),
    displayLine(6,Y,B),
    displayLine(7,Y,B),
    displayLine(8,Y,B),
    displayLine(9,Y,B),
    nl,nl.
```

Estes predicados encarregam-se de imprimir qual a peça do tabuleiro está naquela posição.

```
displaySingleP(P, PiecesY, _) :- member(P,PiecesY),!, write(y), write(' ').
displaySingleP(P, _, PiecesB) :- member(P,PiecesB),!, write(b), write(' ').
displaySingleP(_,_,_) :- write(e), write(' ').
displayPos([H|T],PiecesY,PiecesB) :- displaySingleP(H,PiecesY,PiecesB),
                                     displayPos(T,PiecesY,PiecesB).
displayPos([],_,_).
```

O resto é específico de cada linha. Eis o exemplo de uma dessas linhas:

```
displayLine(1,PiecesY,PiecesB) :- write('    g9--g8--g7--g6    '),
    put_code(186),
    write('          '), displayPos([g9,g8,g7,g6],PiecesY,PiecesB),
    nl,
    write('          '), put_code(92), write(' / '),
    put_code(92), write(' / '), put_code(92), write(' /          '),
    put_code(186),
    write('          '), put_code(92), write('          / '),
    nl.
```

3.3 Lista de Jogadas Válidas

Na implementação do nosso jogo fizemos predicados para definir quais os saltos de peças possíveis, por exemplo estes:

```
validJump(b0, r0, mid).
validJump(y0, g0, mid).
validJump2(X,Y,Z):- validJump(X,Y,Z).
validJump2(X,Y,Z):- validJump(Y,X,Z).
```

Implementamos também os seguintes predicados para obter a lista de todas as jogadas possíveis para qualquer dos jogadores ou para qualquer posição inicial, em que verifica se uma jogada é válida e adiciona-a à lista *Moves*.

```
getAllPossibleMovesAux(Player, [Start|Tail], Yi, Bi, CalcMoves, Moves) :-
    isValid(Player, Yi, Bi, Start, Mid, Final),!,
    getAllPossibleMovesAux(Player, Tail, Yi, Bi, [[Start, Final, Mid]|CalcMoves], Moves).

/* Used when previous condition fails*/
getAllPossibleMovesAux(Player, [_|Tail], Yi, Bi, CalcMoves, Moves):- getAllPossibleMovesAux(Player, Tail, Yi, Bi, CalcMoves, Moves).

getAllPossibleMovesAux(_, [], _, _, Moves, Moves).

getAllPossibleMoves(Player, StartingPositions, YMovers, BMovers, Moves):-
    getAllPossibleMovesAux(Player, StartingPositions, YMovers, BMovers, [], Moves),!.
```

Para uma jogada ser válida vê se cumpre as regras do jogo (ver 3.4).

3.4 Execução de Jogadas

Valida uma jogada através dos predicados seguintes:

```
isValid(b, Yi, Bi, InitialPos, JumpPos, FinalPos) :-
    member(InitialPos, Bi),
    validJump2(InitialPos, FinalPos, JumpPos),
    isEmpty(FinalPos, Yi, Bi),
    isEmpty(JumpPos, Yi, Bi).

isValid(y, Yi, Bi, InitialPos, JumpPos, FinalPos) :-
    member(InitialPos, Yi),
    validJump2(InitialPos, FinalPos, JumpPos),
    isEmpty(FinalPos, Yi, Bi),
    isEmpty(JumpPos, Yi, Bi).

isEmpty(X, Yi, Bi):- boardMembers(L),!, member(X, L), (member(X, Yi) ; member(X, Bi)).

isEmpty(X, Yi, Bi):- boardMembers(L), !,member(X, L),!, \+ member(X, Yi), \+ member(X, Bi).
```

Só depois executa o predicado *move(+MoverPosY, +MoversPosB, +InitialPos, +JumpPos, +FinalPos, -NewMoversPosY, -NewMoversPosB, +TypeofPlayer)*, em que *TypeofPlayer* pode ser *human* ou *bot*, em que elimina a peça da posição onde estava e insere na sua nova posição e eliminando a peça que saltou ou eliminando outra das suas peças dependendo se a peça que saltou era sua ou se era do adversário. Uma parte deste predicado:


```

% Yellow jumps blue mover and selects valid mover to remove.
move(Yi,Bi,InitialPos,JumpPos,FinalPos,Yo,Bo,human) :-
    member(InitialPos,Yi),
    delete(Yi,InitialPos,Yo2),
    append([FinalPos],Yo2,Yo3),
    member(JumpPos,Bi),
    readMoverToRemove(MoverToRemove,Yo3),
    delete(Yo3,MoverToRemove,Yo),
    append([],Bi,Bo).

% Blue jumps blue mover
move(Yi,Bi,InitialPos,JumpPos,FinalPos,Yo,Bo,_) :-
    member(InitialPos,Bi),
    delete(Bi,InitialPos,Bo2),
    append([FinalPos],Bo2,Bo3),
    member(JumpPos,Bi),
    delete(Bo3,JumpPos,Bo),
    append([],Yi,Yo).

```

O nível médio do *bot*, assim como qualquer jogador podem ainda fazer até três saltos consecutivos através dos seguintes predicados:

```

makeConsecutivePlay( +Player, +MoverPosY, +MoversPosB, +FirstInitial,
+PrevFinal, -NewMoversPosY, -NewMoversPosB, +TypeofPlayer, +Count)
readConsecutiveMove( +Player, +MoverPosY, +MoversPosB, +FirstInitial,
+PrevFinal, +Jump, -Final)

```

E também *getConsecutiveMoveForBot* (ver 3.7).

3.5 Avaliação do Tabuleiro

No jogo, através do predicado *points*(ver 3.6) que vê quantos pontos um jogador tem, podemos ver se uma jogada é melhor que outra comparando os pontos dos diferentes tabuleiros.

3.6 Final do Jogo

Verificação do fim do jogo, com identificação do vencedor. Exemplo: *game_over(+Board, -Winner)*.

3.7 Jogada do Computador

Escolha da jogada a efetuar pelo computador, dependendo do nível de dificuldade. Por exemplo: *choose_move(+Level, +Board, -Move)*.

4 Interface com o Utilizador

Descrever o módulo de interface com o utilizador em modo de texto.

5 Conclusões

Que conclui deste projecto? Como poderia melhorar o trabalho desenvolvido?

A Nome do Anexo

Código Prolog implementado devidamente comentado e outros elementos úteis que não sejam essenciais ao relatório.