Trabalho Prático Nº 1 Clobber

Relatório Final

Mestrado Integrado em Engenharia Informática e Computação

Programação em Lógica

Turma 04 Clobber_4:
Luís Oliveira - up201607946
Ricardo Silva - up201607780

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

18 de Novembro de 2018

Conteúdo

1 Introdução

No âmbito da unidade curricular de Programação em Lógica, do curso Mestrado Integrado em Engenharia Informática e Computação, tivemos que elaborar um jogo em PROLOG. Esse jogo foi selecionado pelo grupo dentro de um leque de várias opções que nos foram disponibilizados pelos docentes.

Escolhemos o Clobber devido à sua jogabilidade simples e a combinação de jogadas possíveis. Apesar de simples, tal como no jogo das Damas, é possível haver uma boa prática mental e estratégica com o desenrolar de uma partida entre dois elementos.

O objetivo deste trabalho foi a aplicação dos primeiros conceitos interiorizados nas aulas teóricas e desenvolvidos nas aulas práticas da unidade curricular. Nas teóricas podemos ouvir os conceitos e nas teórico-praticas implementa-los mas é num projeto prático que verdadeiramente aprendemos e cimentamos o conhecimento

Este método de avaliação torna-se importante pois permite-nos avaliar os conhecimentos que adquirimos até então e saber se somos ou não capazes de, com uma linguagem de programação nova e um paradigma completamente diferente do que estamos habituados, produzir algo de útil para o quotidiano e futuro.
Este relatório encontra-se dividido em várias secções tais como:

1. Introdução, onde se descreve os objetivos do trabalho.
2. Descrição sucinta do jogo Clobber, a sua história e, principalmente, as suas regras.
3. Lógica do Jogo, com Descrição do projeto e implementação da lógica do jogo em Prolog
4. Conclusões que tiramos deste projeto e como poderíamos melhorar o trabalho desenvolvido
5. Bibliografia consultada

2 O Jogo Clobber

2.1 História

Inventado em 2001, por Michael H. Albert, J.P. Grossman and Richard Nowakowski, Clobber é um jogo jogado, usualmente, num tabuleiro 5 colunas por 6 linhas e constituído por dois tipos de peças: pretas e brancas.

Uma exemplificação de um jogo pode ser visualizado no seguinte link:

`https://www.youtube.com/watch?v=H3zSzqur_II`

2.2 Regras

2.2.1 Início

As peças são distribuídas por todo o tabuleiro da seguinte forma: peças pretas ficam nas casas pretas e as brancas ficam nas casas brancas. Começam as brancas.

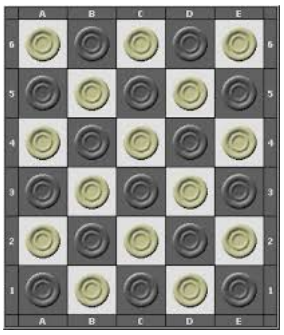

Figura 1: Distribuição Inicial das peças

2.2.2 Turnos

A cada turno um jogador move uma das suas peças num sentido ortogonal (Norte-Sul-Oeste-Este) em direção a uma casa do tabuleiro que contenha uma peça de cor contrária. A peça de cor contrária é removida do tabuleiro e a peça do jogador ocupa a nova casa de onde foi retirada a peça do adversário.[1]

2.2.3 Vencedor e fim do jogo

O vencedor é aquele que fizer o último movimento, ou seja, que move a última peça, por outras palavras, o jogador que numa jogada não se conseguir mover perde o jogo.[2]

[1] https://en.wikipedia.org/wiki/Clobber Info disponível a 15/10/2018
[2] https://www.sciencenews.org/article/getting-clobbered

3 Lógica do Jogo

Descrever(não basta copiar o código fonte) o projeto e implementação da lógica do jogo em Prolog, incluindo a forma de representação do estado do tabuleiro e sua visualização, geração de jogadas válidas, execução de jogadas, determinação do final do jogo, avaliação do tabuleiro e cálculo das jogadas a realizar pelo computador utilizando diversos níveis de jogo. O predicado de início de jogo deve chamar-se play().

Correr o programa:

No prompt do Prolog fazer

```
consult('clobberInit.pl').
```

De seguida chamar o predicado

```
play.
```

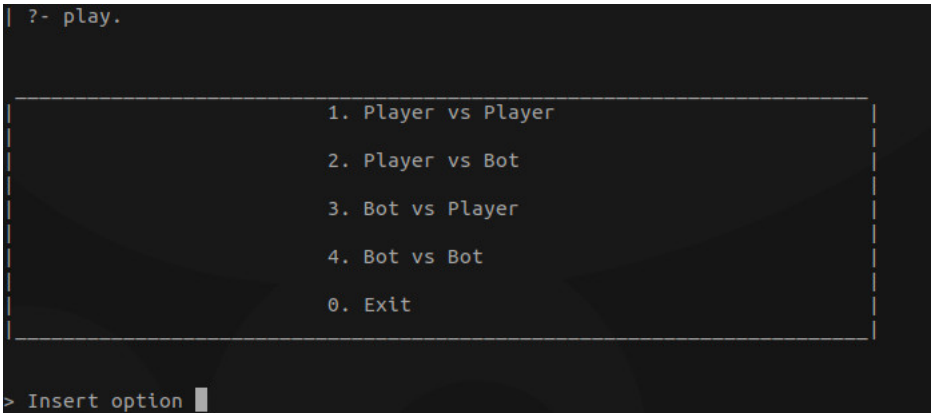

Figura 2: Menu Principal

Temos assim 4 possibilidades no menu principal: Humano contra outro Humano, Humano contra Bot e vice-versa ou Bot contra Bot.

3.1 Representação do Estado do Jogo

A nível interno o tabuleiro será representado por uma estrutura de dados correspondente a uma lista de listas. Desta forma, cada célula poderá tomar os valores de *empty*, *black* ou *white*.

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

symbol(empty,S) :- S='.'.
symbol(black,S) :- char_code(S,9863).
symbol(white,S) :- char_code(S,9865).
```

3.2 Visualização do Tabuleiro

Para a visualização do tabuleiro os valores serão substituídos por . , ⚇ *e* ⚉.
Desta substituição traduz-se para estados ilustrativos que se seguem:

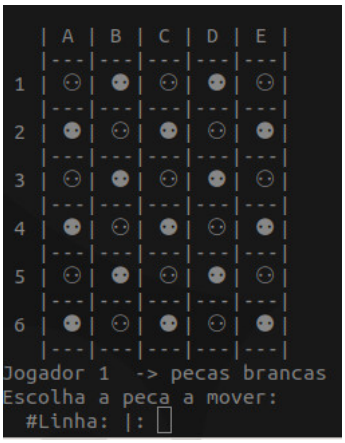

Figura 3: Estado Inicial - inicio do jogo

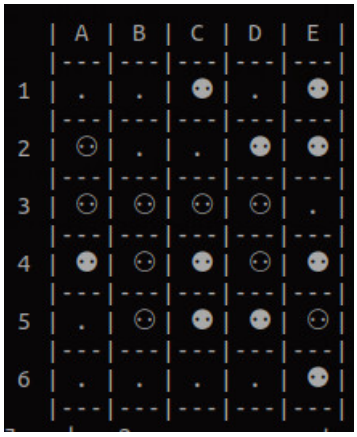

Figura 4: Um possível Estado Intermédio

Através da observação da Figura 4, a estratégia de jogo reside no isolamento de peças do adversário, e assim, impossibilitando-o de mover-se.

Uma nota na Figura 5 é essencial. Ao efetuarem a última jogada [de 2-e para 2-d], ou seja, o único movimento em que era possível avançar para uma casa com uma peça contrária, o vencedor são as peças brancas (2), em virtude de o jogador de peças pretas (1) não poder efetuar mais nenhuma jogada.

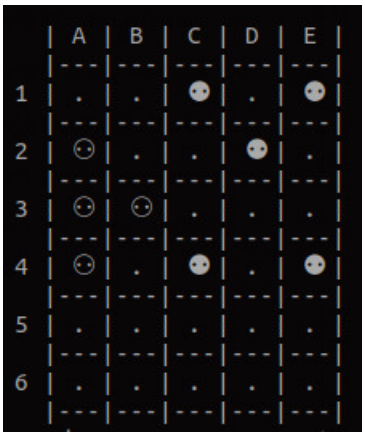

Figura 5: Um possível Estado Final

3.3 Lista de Jogadas Válidas

Para a obtenção de uma lista de todas as jogadas possíveis válidas para a peca numa dada célula com uma peça controlada pelo Bot implementamos o predicado valid_moves/3.

Este usa o findAll que vai verificar o objectivo seleccionarBotJogada/7 que tenta sequencialmente jogadas para o Sul, Norte, Oeste, Este até uma resultar.

```
/**
 * valid_moves(+Tabuleiro,+LineIndex,+ColumnIndex,+ColorPlayer,
 -ListasJogadas)
 * @brief Devolve lista com todas as jogadas possiveis validas para
 a peca na celula dada por linha e coluna
 * @param +Tabuleiro: tabuleiro actual
 * @param +LineIndex: indice da linha
 * @param +ColumnIndex: indice da coluna
 * @param +ColorPlayer: cor da peca
 * @param -ListasJogadas: lista com todas as jogadas possiveis validas
 para a peca na celula dada por linha e coluna
 */
valid_moves(TabuleiroInicial, LineIndex, ColumnIndex, ColorPlayer,
ListasJogadas):-
    findall([NewLineIndex-NewColumnIndex],
    seleccionarBotJogada(TabuleiroInicial, LineIndex, ColumnIndex,
    NewLineIndex,      NewColumnIndex, ColorPlayer), ListasJogadas).
```

3.4 Execução de Jogadas

A execução de uma jogada no tabuleiro é efetuada através do predicado move/7. O novo estado é devolvido através de -TabuleiroFinal, já validado. O movimento só é feito no caso de se ter pedido que fosse para uma casa vizinha e onde estivesse uma peca adversária. A parte da verificação de qual peça na casa de destino é feita através de unificações, enquanto que a verificação de ser uma casa vizinha é feita aquando da introdução do pedido.

Se a jogada for das brancas, a primeira versão sucede, caso seja das pretas falha e sucede a segunda versão.

```
/**
* move(+TabuleiroInicial, +RowIndex, +ColumnIndex, +PP_RowIndex,
+PP_ColumnIndex, -TabuleiroFinal, +Color)
* @brief Validar e executar a jogada das pecas brancas
* @param +TabuleiroInicial -> board status before the move
* @param +RowIndex -> linha da peca selecionada que vai se movimentar
* @param +ColumnIndex -> coluna da peca selecionada que vai se movimentar
* @param +PP_RowIndex -> Linha da P.roxima P.osicao
* @param +PP_ColumnIndex -> Coluna da P.roxima P.osicao
* @param +Color -> black or white : which piece to move
* @param -TabuleiroFinal -> board status after the move
*/
move(TabuleiroInicial, RowIndex,ColumnIndex,PP_RowIndex,PP_ColumnIndex,
TabuleiroFinal, Color):-
    getValueFromMatrix(TabuleiroInicial, RowIndex, ColumnIndex,
    ValueJogador),
    ValueJogador == Color,
    write('Peca escolhida valida\n'),
    getValueFromMatrix(TabuleiroInicial, PP_RowIndex, PP_ColumnIndex,
    ValueAdversario),
    ValueJogador == white ,
    !,
    ValueAdversario == black,
    write('Jogada Valida'),
    replaceInMatrix(TabuleiroInicial, PP_RowIndex, PP_ColumnIndex, Color,
    TabuleiroNovo),
    replaceInMatrix(TabuleiroNovo, RowIndex, ColumnIndex, empty,
    TabuleiroFinal).

/*
* @brief (vide anterior) validar a jogada das pecas pretas
*/
move(TabuleiroInicial, RowIndex,ColumnIndex,PP_RowIndex,PP_ColumnIndex,
TabuleiroFinal, Color):-
    getValueFromMatrix(TabuleiroInicial, PP_RowIndex, PP_ColumnIndex,
    ValueAdversario),
    ValueAdversario == white,
    write('Jogada Valida'),
    replaceInMatrix(TabuleiroInicial, PP_RowIndex, PP_ColumnIndex, Color,
    TabuleiroNovo),
    replaceInMatrix(TabuleiroNovo, RowIndex, ColumnIndex, empty,
    TabuleiroFinal).
```

3.5 Final do Jogo

A verificação do fim do jogo é feita com o predicado gameOver/2, que chama anunciamento/1 para indicar quem perdeu.

Para poder verificar se estamos numa situação de fim de jogo o gameOver chama os predicados posicoesPecasNoTabuleiro/3 e loop/4.

O primeiro vai descobrir as posições de todas as pecas da cor que vai jogar a seguir e a segunda pega nessa informação e percorre-as a todas para achar o numero total de jogadas validas do jogador.

Se o número de jogadas possíveis das peças da cor que vai jogar a seguir for zero então o jogo acabou e perdeu.

```
/**
 * @brief condicao de terminacao do jogo. se lista de jogadas possiveis
 de todas as pecas de uma cor for vazia esse player ja nao joga e
 perdeu o jogo
 * gameOver(+Tabuleiro, -Looser)
 * @param -Looser: Color do jogador que perde
 * @param Tabuleiro: tabuleiro actual
*/
gameOver(Tabuleiro, Looser):-
        Looser == black,
        !,
        CorContraria = white,
        posicoesPecasNoTabuleiro(Tabuleiro, Looser,ListaDePecasNoTabuleiro),
        loop(Tabuleiro, CorContraria, ListaDePecasNoTabuleiro, Total),
        Total == 0,
        !,
        anunciamento(Looser).

gameOver(Tabuleiro, Looser):-
        CorContraria = black,
        posicoesPecasNoTabuleiro(Tabuleiro, Looser,ListaDePecasNoTabuleiro),
        loop(Tabuleiro, CorContraria, ListaDePecasNoTabuleiro, Total),
        Total == 0,
        !,
        anunciamento(Looser).

/**
 * posicoesPecasNoTabuleiro(+TabuleiroInicial,+Color,-ListaDePares)
 * @brief Devolve todas as posicoes das pecas de Color actualmente
 existentes no tabuleiro
 * @param TabuleiroInicial: tabuleiro actual
 * @param Color: cor da peca
 * @param ListaDePares: posicoes de todas as pecas de Color
 actualmente no tabuleiro
 * Devolve todas as posicoes das pecas no tabuleiro
*/
posicoesPecasNoTabuleiro(TabuleiroInicial,Color,ListaDePares):-
    findall([LineIndex-ColumnIndex], selecionarPecaForBot(TabuleiroInicial,
    [LineIndex-ColumnIndex], Color), ListaDePares).
```

```
/**
 * loop(+Tabuleiro, +CorContraria, +Lista, -Total).
 * @brief descobre o numero total de jogadas validas do adversario
 * @param +Tabuleiro: tabuleiro actual
 * @param +CorContraria: cor do adversario
 * @param +Lista: celulas onde tem pecas
 * @param -Total: total de jogadas validas do adversario
 */
loop(_,_,[],0).
loop(Tabuleiro, CorContraria, [[Line-Column]|Tail], Total):-
        jogadasValidasPorPeca(Tabuleiro, Line, Column, CorContraria,
        ListaDePares),
        length(ListaDePares, Tamanho),
        loop(Tabuleiro, CorContraria, Tail, AuxTotal),
        Total is (AuxTotal + Tamanho).
```

3.6 Avaliação do Tabuleiro

Forma(s) de avaliação do estado do jogo.
O predicado deve chamar-se value(+Board, +Player, -Value).

3.7 Jogada do Computador

A escolha da jogada a efetuar pelo computador é feita com predicado choose_move/3. Dependendo do nível de dificuldade chama um de outros dois:

1. Aleatório: jogarLeBot/2
2. Inteligente:

O modo aleatório gera um valor random entre 1 e o número de peças com jogadas validas para escolher qual peça jogar e de seguida outro random que vai até ao número de jogadas válidas dessa peça para escolher a direcção.

O modo Inteligente usa uma heurística...

```
/**
 * choose_move(+Tabuleiro, -TabuleiroFinal, +Nivel)
 * @brief Escolhe o movimento do Bot consoante o nivel escolhido
 * @param +Tabuleiro: tabuleiro actual
 * @param -TabuleiroFinal: tabuleiro futuro
 * @param +Nivel: nivel escolhido para o Bot  (A=Aleatorio; I=Inteligente)
 */
choose_move(Tabuleiro, TabuleiroFinal, Nivel):-
    Nivel=='A',!,%aleatorio
    jogarLeBot(Tabuleiro, TabuleiroFinal).

choose_move(Tabuleiro, TabuleiroFinal, _Nivel):-
    jogarLeBot(Tabuleiro, TabuleiroFinal). %MUDAR PARA FUNCAO INTELIGENTE


/**
```

```
 * jogarLeBot(+TabuleiroInicial, -TabuleiroFinal)
 * @brief Generates a random play for the bot without being clever
 - a black piece eats a white one
 * @param +TabuleiroInicial: tabuleiro actual
 * @param -TabuleiroFinal: tabuleiro futuro
*/
jogarLeBot(Tabuleiro, TabuleiroFinal):-
    posicoesPecasNoTabuleiro(Tabuleiro,black,ListaDePares),
        escolha(Tabuleiro, ListaDePares,ListaParaLimpar),
    cleanLista(ListaParaLimpar, _NovaLista),
   direccaoDaJogada(Tabuleiro,[Line-Column],white, _ListaJogadasVizinhas),
    replaceInMatrix(_TabuleiroInicial, NewLineIndex, NewColumnIndex,
    black, TabuleiroNovo),
    replaceInMatrix(TabuleiroNovo, Line, Column, empty, TabuleiroFinal),.
jogarLeBot(TabuleiroInicial, TabuleiroFinal):-
    jogarLeBot(TabuleiroInicial, TabuleiroFinal).
```

4 Conclusões

Tendo em conta os objetivos propostos, é de referir que as opções 3 e 4 do Menu de jogo não estão a funcionar. A opção 3 - **Bot vs Player** não foi implementada com sucesso, porque apesar de considerarmos o código manipulável o suficiente para reaproveitar as funções, deparamos com um erro que não foi possível corrigir em tempo útil. No nosso ponto de vista bastaria mudar os atómos ("white"e "black") presentes no *choose_move* para uma variável ou mesmo aplicar o seguinte código no inicio do *choose_move*:

((ColorPlayer == black, ColorContraria = white); ColorPlayer == white, ColorContraria = black))

mas, deparamos com um erro que possivelmente advém da forma lógica de uma função de baixo nível.

A escolha do nível de dificuldade, que coincide com a opção 3 e 4 do Menu, não está disponível pois não se implementou em tempo útil o *choose_move* com um value de escolha de melhor caminho, no entanto, a ideia do esboço apresentado no ficheiro *ai.pl* seria procurar o caminho na árvore que maximiza-se o número de jogadas possíveis, porque afinal, o Clobber perde quem não tem jogadas. Salienta-se que o *Choose_move* implementado apenas está sobre a forma de random sob a lista de jogadas possiveis e válidas.

Mesmo assim, o jogo Clobber exigiu imenso tempo ao grupo para a sua implementação. Uma linguagem e um paradigma novos que inicialmente tornam as iterações de desenvolvimento demasiado longas

Consideramos que o resultado final obtido é muito positivo e que os conhecimentos adquiridos durante o desenvolvimento do projeto foram inestimáveis. Apesar de várias vicissitudes e de inevitáveis colisões com trabalhos de outras unidades curriculares, conseguimos concluir o que havíamos planeado.

Pela análise do código, consegue-se distinguir claramente, um incremental melhoramento na manipulação e escrita de funções. Somos unânimes em afirmar que para o estado atual do jogo, conseguíamos escrever um outro programa com muito menos linhas e mais eficiente.

O projeto apresentou-se como um desafio que, com esforço, dedicação e empenho conseguimos ultrapassar, retirando experiência e conhecimento do mesmo e tornar o Clobber num jogo muito apelativo que proporciona ao jogador uma boa prática mental.

As dificuldades encontradas foram sendo superadas, porém poderiam haver melhorias, nomeadamente se tivéssemos iniciado o projeto já a pensar em requisitos e funções que não sabíamos ter que desenvolver mais tarde. Ale disso devíamos ter conseguido um melhor código e funções mais eficientes.

Em suma, o grupo gostou da experiência de desenvolvimento de um jogo na linguagem PROLOG. Ao contrário do que estamos habituados, este tipo de linguagem requer um pensamento lógico em cada predicado desenvolvido.

5 Bibliografia

Bratko, Ivan ; 2000 ; Prolog Programming for Artificial Intelligence 3/E ; University of Ljubljana ; Addison-Wesley ; ISBN-10: 9780201403756 • ISBN-13: 978-0201403756

Clocksin W. F. and Mellish C. S. ; 1984 ; Programming in prolog ; A Springer-Verlag Telos ; ISBN 10: 3540110461 • ISBN 13: 9783540110460

Sterling, Leon and Shapiro, Ehud; 1994 ; The Art of Prolog 2/E; Mit Pr. ; ISBN 10: 0262691639 • ISBN 13: 9780262691635

Torres, Delfim F. M. ; 2000 ; Introdução à Programação em Lógica. ed. 1 ; Aveiro: Universidade de Aveiro ; ISBN: 972-8021-93-3.

https://sicstus.sics.se/

http://www.swi-prolog.org/

https://boardgamegeek.com/boardgame/23864/clobber

https://en.wikipedia.org/wiki/Clobber

https://www.chessprogramming.org/Clobber

[FIM]

6 ANEXOS

6.1 Anexo 1

Código-fonte anexado no ficheiro zip entregue.