

# Aplicação em Prolog para um Jogo de Tabuleiro

Relatório Final



Mestrado Integrado em Engenharia Informática e  
Computação

Programação em Lógica

**Pivit\_2:**

João Fernando de Sousa Almeida - up201006563

Filipe Manuel Ferreira Cordeiro – up200105009

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

12 de Setembro de 2014

## Resumo

Este relatório diz respeito ao trabalho prático desenvolvido para a unidade curricular de PLOG com o objetivo de desenvolver uma aplicação para jogar um jogo de tabuleiro, usando Prolog como linguagem de implementação. O jogo em causa é o Pivit, um jogo para 2 a 4 jogadores jogado num tabuleiro de xadrez com peças especiais. As peças possuem dois lados diferentes que representam os dois tipos de peça existentes (minions e masters) e podem-se deslocar ao longo de um linha (vertical e horizontal) do tabuleiro de acordo com um conjunto de regras especiais. As regras e movimentos das peças são bastante simples mas permitem uma estratégia profunda e complexa tal como no xadrez (salvo a devida comparação).

Apesar de os requisitos do trabalho apenas preverem um jogo para dois jogadores, considerando que o jogo prevê até quatro jogadores e que a implementação não seria mais complexa, optou-se por permitir jogos com dois, três ou quatro jogadores. O jogo permite qualquer combinação de modos de utilização dos jogadores, i.e., podem ser todos manuseados por pessoas (tipo Humano), todos pelo Computador, ou um misto. Deveriam ser previstos dois níveis de jogo para o computador mas tendo em conta a complexidade da estratégia e da dificuldade em escolher a melhor jogada de forma clara (à imagem do xadrez) optou-se por desenvolver apenas um nível inteligência artificial que consiste em jogar aleatoriamente dando prioridade a alguns movimentos específicos se estes forem possíveis.

Este trabalho, para além da familiarização com a linguagem Prolog, permitiu perceber as potencialidades da programação em lógica, nomeadamente do backtracking, e como este paradigma se adequa melhor a determinado tipo de problemas.

# Índice

1	Introdução .....	4
2	O Jogo <i>Pivit</i> .....	5
3	Lógica do Jogo .....	7
3.1	Representação do Estado do Jogo .....	7
3.2	Visualização do Tabuleiro .....	8
3.3	Lista de Jogadas Válidas .....	12
3.4	Execução de Jogadas.....	14
3.5	Avaliação do Tabuleiro e Jogada do Computador .....	15
3.6	Final do Jogo.....	17
4	Interface com o Utilizador .....	19
5	Conclusões .....	21
	Bibliografia .....	22
	Anexos .....	23

# 1 Introdução

Este trabalho prático tinha como objetivo implementar um jogo de tabuleiro na linguagem de programação em lógica Prolog. Dentro das opções de jogos de tabuleiro disponibilizadas, a nossa escolha recaiu sobre *Pivit* que captou a nossa atenção por ser um jogo moderno e com regras fáceis de compreender, mas que simultaneamente permite a elaboração de estratégias bastante profundas.

Este relatório foi estruturado com base nas indicações fornecidas no enunciado do Trabalho Prático e no relatório intercalar. Apresenta-se o trabalho e o jogo em questão; elabora-se sobre os vários aspetos da lógica do jogo: da representação do estado até às jogadas do computador; destaca-se a interface com o utilizador e apresentam-se as conclusões.

## 2 O Jogo Pivit

Pivit é um jogo de estratégia abstrato para 2 a 4 jogadores, baseado num tabuleiro de xadrez e criado por Tyler Neylon no final de 2012. Para obter financiamento para desenvolver Pivit, Neylon recorreu ao *site* de *crowdfunding* Kickstarter, mas não conseguiu atingir o valor pretendido, pelo que o jogo nunca chegou a ser comercializado.

Visto que Pivit se baseia num tabuleiro de xadrez, necessita apenas de algumas peças, especialmente desenhadas para o jogo por Neylon, para ser jogado. Estas peças podem facilmente ser improvisadas e o próprio autor disponibiliza no *site* oficial do jogo uma versão “*Print and Play*”. Na Figura 2.1 ilustra-se alguns tabuleiros iniciais típicos no Pivit, sendo a representação inferior direita o esquema mais simples e rápido (tabuleiro menor e apenas dois jogadores), recomendado para principiantes e o tabuleiro superior direito o mais complexo (tabuleiro 8x8 com quatro jogadores).

As peças possuem uma cor que representa o respetivo jogador e possuem dois lados diferentes, sendo que um lado representa os *minions* (as peças básicas iniciais), e o outro, os *masters* (Figura 2.2). Ambos os lados possuem setas de direção. As peças são dispostas nos quadrados da periferia do tabuleiro - exceto nos cantos - de forma pré-definida dependente do número de jogadores.

Tal como ilustrado na Figura 2.3, no jogo, os jogadores movem, à vez e sempre na mesma ordem (no caso de mais de 2 jogadores), as suas peças no tabuleiro. Cada peça desloca-se de uma casa (quadrado do tabuleiro) para outra ao longo de uma linha ou de uma coluna (dependendo da direção das setas da peça) mas nunca na diagonal. Cada vez que uma peça se desloca esta roda 90°, alterando dessa forma a direção do seu próximo movimento. Os *minions* não se podem deslocar para casas do tabuleiro com a mesma cor da sua casa inicial, i.e., se a sua casa inicial é branca têm que se deslocar para uma casa preta e vice-versa (na prática têm de se deslocar um número de casas ímpar). Os *masters* não estão sujeitos a esta regra, i.e., podem-se deslocar para casas da mesma cor da casa inicial na mesma linha ou coluna (dependendo da sua orientação). Aliás, esta é a única diferença entre os dois tipos de peça. Para ser promovido a *master*, um *minion* tem que se deslocar para um canto do tabuleiro (nesse momento a peça é virada ao contrário, para indicar o seu novo tipo, mas mantendo a direção) (Figura 2.4). Uma peça (*minion* ou *master*) não pode saltar sobre outra, quer seja ela do mesmo jogador ou do adversário, i.e., uma peça não se pode mover de uma casa para outra se entre estas casas houver outras peças. Uma peça pode, no entanto, deslocar-se para a mesma posição de uma peça de um jogador adversário (sem saltar sobre outras), removendo-a do jogo.

O jogo termina quando não houver mais *minions* no tabuleiro e o vencedor é o jogador que tiver mais *masters*. Em

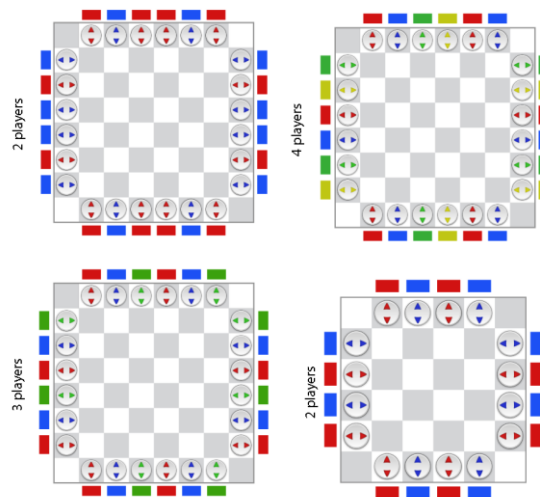


Figura 2.1 – Exemplos de Tabuleiros



Fig. 2.2 – Representação das peças  
(adaptado do site oficial do jogo)

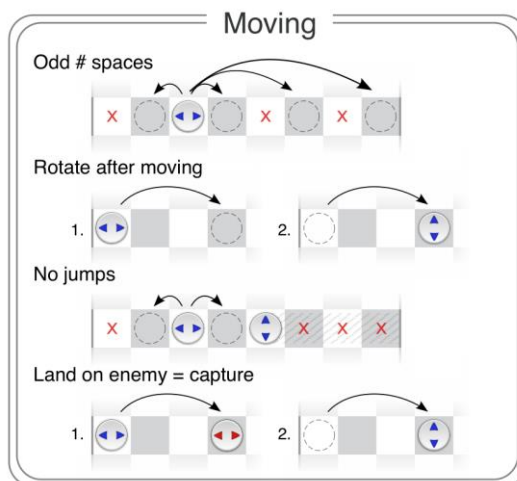


Figura 2.3 - Movimentos

caso de empate vence o jogador que primeiro promoveu um *minion* para *master*.

Se um jogador não tiver nenhuma peça é eliminado. Se só restar um jogador, este é considerado o vencedor. Na figura 2.5 são apresentadas de forma gráfica as condições de vitória.

Se um jogador não conseguir mover nenhuma peça, passa a sua vez ao jogador seguinte.

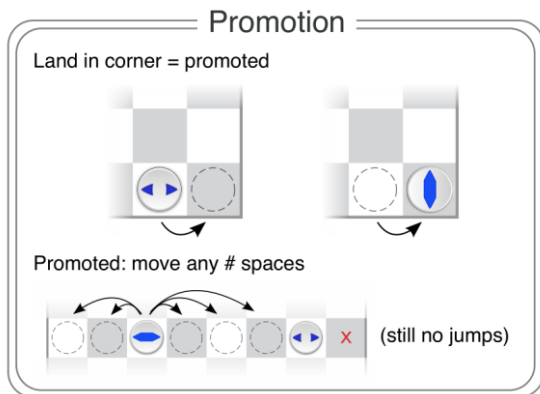


Figura 2.4 – Promoção  
(adaptado do site oficial)

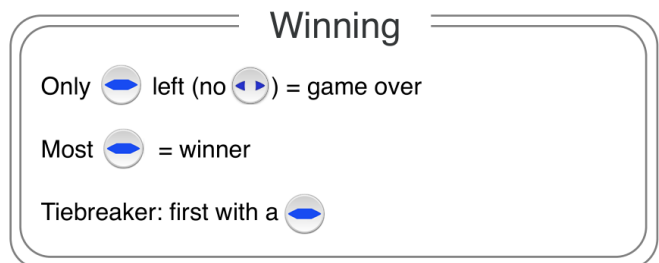


Figura 2.5 – Final do Jogo  
(adaptado do site oficial)

É possível ver um pequeno exemplo de um início de jogo, na Figura 2.6

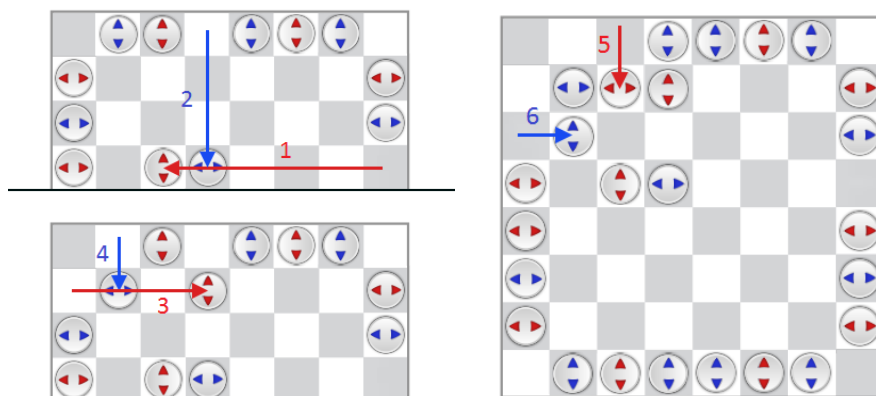


Figura 2.6 – Exemplo de um jogo de 2 jogadores, começando o jogador vermelho  
(adaptado do site BoardGameGeek)

### 3 Lógica do Jogo

#### 3.1 Representação do Estado do Jogo

Para a representação do estado do tabuleiro optou-se pela utilização de uma lista com 8 elementos, em que cada um desses elementos representa uma linha do tabuleiro, e é também uma lista de 8 elementos. Por sua vez cada um dos elementos desta lista representa um quadrado dessa linha e é também uma lista: uma lista vazia no caso de não haver nenhuma peça nesse quadrado; ou uma lista de 3 elementos caso exista uma peça. Os 3 elementos representam informação da peça existente: o primeiro elemento identifica o jogador a quem pertence a peça (1, 2, 3 ou 4); o segundo identifica se a peça é *minion* ou *master* (m ou 'M'); e o terceiro identifica a orientação da peça (h para horizontal e v para vertical).

Deste modo o estado do tabuleiro no início de um jogo para 3 jogadores (ilustrado na figura 3.1.1) seria representado em Prolog pela seguinte lista:

```
[
  [ [], [1, m, v], [2, m, v], [3, m, v], [1, m, v], [2, m, v], [3, m, v], [] ],
  [ [1, m, h], [], [], [], [], [], [], [1, m, h] ],
  [ [2, m, h], [], [], [], [], [], [], [2, m, h] ],
  [ [3, m, h], [], [], [], [], [], [], [3, m, h] ],
  [ [1, m, h], [], [], [], [], [], [], [1, m, h] ],
  [ [2, m, h], [], [], [], [], [], [], [2, m, h] ],
  [ [3, m, h], [], [], [], [], [], [], [3, m, h] ],
  [ [], [1, m, v], [2, m, v], [3, m, v], [1, m, v], [2, m, v], [3, m, v], [] ]
]
```

O estado do tabuleiro ilustrado na figura 3.1.2 (jogo com 4 jogadores) seria representado em Prolog pela seguinte lista:

```
[
  [ [], [], [], [], [2, 'M', v], [], [], [] ],
  [ [], [], [], [], [], [], [], [3, 'M', v] ],
  [ [], [], [], [], [], [], [], [] ],
  [ [1, 'M', h], [], [], [], [4, m, h], [], [], [4, m, h] ],
  [ [], [], [], [], [], [], [], [] ],
  [ [], [], [], [], [], [], [], [] ],
  [ [], [], [], [], [], [], [], [] ],
  [ [], [], [], [], [], [], [], [] ]
]
```

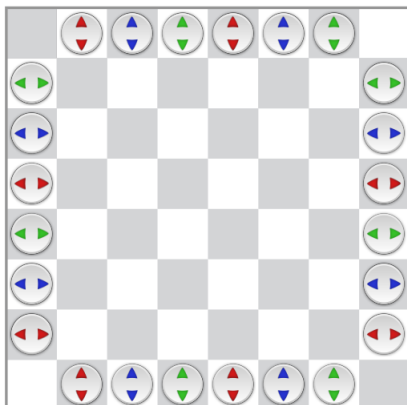


Fig. 3.1.1 – Posições iniciais num jogo para 3 jogadores.

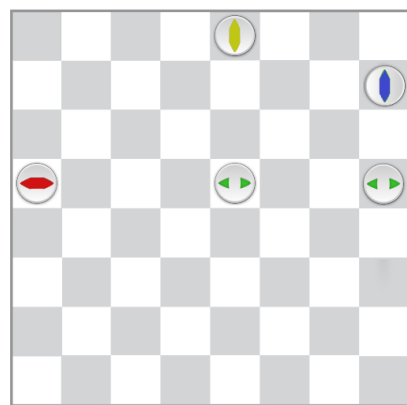


Fig. 3.1.2 – Jogo para 4 jogadores prestes a terminar.

## 3.2 Visualização do Tabuleiro

Para visualizar o estado do tabuleiro foram implementados vários predicados. Em primeiro lugar tem-se o predicado *printboard* que tem 2 argumentos. O primeiro argumento é a lista que representa o tabuleiro, e o segundo a cor do primeiro quadrado do tabuleiro em que 'w' corresponde a um quadrado branco e 'b' a um quadrado preto. O segundo argumento é utilizado devido à natureza alternada dos quadrados de um tabuleiro de xadrez e para ser possível usar a recursividade neste predicado. *Printboard* imprime no ecrã vários caracteres "-" para representar o limite superior das linhas, de seguida imprime os números das linhas do tabuleiro utiliza os predicados *printlinetop* e *printlinebottom* sobre o primeiro elemento da lista (correspondente a uma linha) e por último utiliza recursivamente o predicado *printboard* sobre a cauda da lista alterando o argumento da cor (de 'w' para 'b' ou vice-versa), visto que a linha seguinte vai começar com a cor contrária à da linha actual. No caso de chegar ao fim da lista (argumento é uma lista vazia), imprime novamente no ecrã vários caracteres "-" para representar o limite inferior do tabuleiro, assim como as letras correspondentes às 8 colunas (de A a H).

A informação de cada quadrado é apresentada em 3 linhas diferentes por isso são utilizados os predicados *printlinetop*, *printlinemiddle* e *printlinebottom* que funcionam de maneira idêntica mas apresentam informação das linhas superior, intermédia e inferior, respetivamente. Ambos os predicados têm 2 argumentos: o primeiro é uma lista que representa a linha; o segundo identifica a cor do primeiro quadrado da linha (0 ou 1 tal como em *printboard*). O funcionamento é semelhante ao de *printboard*: utilizam os predicados *printsquaretop*, *printsquaremiddle* ou *printsquarebottom* (dependendo se é *printlinetop*, *printlinemiddle* ou *printlinebottom*, respetivamente) sobre o primeiro elemento da lista (correspondente a um quadrado neste caso) e utilizam recursividade sobre a cauda da lista alterando o argumento da cor. No caso de chegarem ao fim da lista (argumento é uma lista vazia), imprimem no ecrã o carácter "|" para representar o limite direito do tabuleiro.

O predicado *printsquaretop* imprime '^' no caso de ser uma peça de orientação vertical, correspondendo ao carácter 'v' imprimido através de *printsquarebottom*. De forma equivalente, *printsquaremiddle* imprime os caracteres '<' e '>' caso a peça tenha orientação vertical, assim como a letra correspondente ao jogador (maiúscula no caso de ser um *master*), obtida a partir do predicado auxiliar *player*. Caso se trate dum quadrado vazio, então são impressos apenas espaços em branco ou o quadrado é preenchido com caracteres '/' para preencher o fundo dum quadrado preto – no caso de ser um quadrado preto com uma peça, os caracteres restantes do quadrado são preenchidos com os mesmos caracteres, para representar o fundo. // TODO MIDDLE

O primeiro parâmetro destes predicados corresponde ao quadrado a ser impresso, e o segundo à cor do fundo do quadrado em questão.

A implementação dos predicados construídos para visualização dos estados do tabuleiro é apresentada a seguir.

```
display_game(Board):- printboard(Board,w).
```

```
printboard([C|R],w):-
    write(' -----'),
    nl,
    write(' '),
    printlinetop(C,w),
    nl,
    length([C|R],RW),
    write(RW),
    write(' '),
    printlinemiddle(C,w),
    nl,
    write(' '),
    printlinebottom(C,w),
    nl,
    printboard(R,b).
```



```

printboard([C|R],b):-
    write(' -----'),
    nl,
    write(' '),
    printlinetop(C,b),
    nl,
    length([C|R],RW),
    write(RW),
    write(' '),
    printlinemiddle(C,b),
    nl,
    write(' '),
    printlinebottom(C,b),
    nl,
    printboard(R,w).

```

```

printboard([], _):-
    write(' -----'),
    nl,
    write('  A  B  C  D  E  F  G  H'),
    nl,
    nl.

```

```

printlinetop([C|R],w):-
    printsquaretop(C,w),
    printlinetop(R,b).

```

```

printlinetop([C|R],b):-
    printsquaretop(C,b),
    printlinetop(R,w).

```

```

printlinetop([], _):-
    write('').

```

```

printlinemiddle([C|R],w):-
    printsquaremiddle(C,w),
    printlinemiddle(R,b).

```

```

printlinemiddle([C|R],b):-
    printsquaremiddle(C,b),
    printlinemiddle(R,w).

```

```

printlinemiddle([], _):-
    write('').

```

```

printlinebottom([C|R],w):-
    printsquarebottom(C,w),
    printlinebottom(R,b).

```

```

printlinebottom([C|R],b):-
    printsquarebottom(C,b),

```

```

        printlinebottom(R,w).
printlinebottom([],_):-
    write('|').

printsquaretop([_,_,v],w):-
    write('| ^ ').
printsquaretop([_,_,v],b):-
    write('|//^//').
printsquaretop( _, w):-
    write('| ').
printsquaretop( _, b):-
    write('|////').

printsquaremiddle([P,M,h],_):-
    write('< '),
        player(P,M,Letter),
        write(Letter),
        write(' > ').
printsquaremiddle([P,M,v],w):-
    write('| '),
        player(P,M,Letter),
        write(Letter),
        write(' ').
printsquaremiddle([P,M,v],b):-
    write('|//'),
        player(P,M,Letter),
        write(Letter),
        write('//').
printsquaremiddle([],w):-
    write('| ').
printsquaremiddle([],b):-
    write('|////').

printsquarebottom([_,_,v],w):-
    write('| v ').
printsquarebottom([_,_,v],b):-
    write('|//v//').
printsquarebottom( _, w):-
    write('| ').
printsquarebottom( _, b):-
    write('|////').

player(1,m,a).
player(1,'M','A').

```

```

player(2,m,b).
player(2,'M','B').
player(3,m,c).
player(3,'M','C').
player(4,m,d).
player(4,'M','D').

```

A figura 3.2.1 corresponde aos *outputs* produzidos pelo predicado de visualização para os estados do tabuleiro representados anteriormente nas figuras 3.1.1 e 3.1.2 respetivamente.

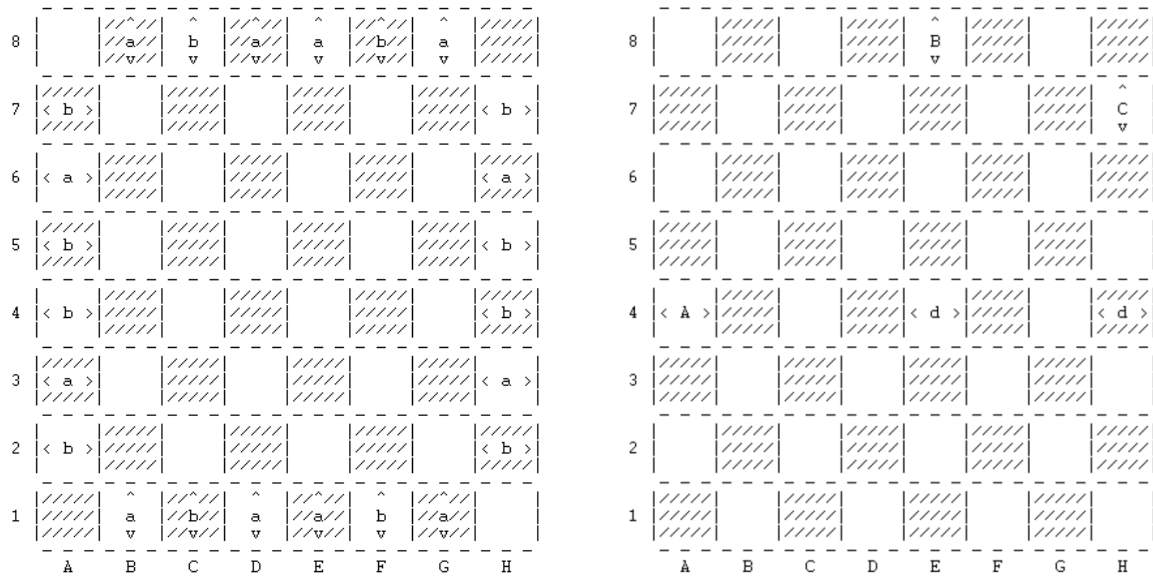


Fig. 4.2.1 – *Outputs* do Sicstus produzidos pelo predicado de visualização

### 3.3 Lista de Jogadas Válidas

É utilizado o predicado *valid\_move* para verificar se são cumpridas todas as restrições de movimentos utilizando diversos predicados auxiliares. No caso de *input* de utilizador (*valid\_move/8*) não falha mas imprime a primeira restrição que não cumpre e requer novo input de valores. O predicado *inbounds* verifica se as coordenadas são legais. É utilizado o predicado *piece* para verificar se a peça pertence ao jogador de quem é a vez de jogar e se sim, qual a sua orientação. O predicado *odd* verifica se o número de movimentos é ímpar no caso de a peça ser *minion*. O predicado *position* permite obter as coordenadas finais de um movimento e verificar se estas são legais com base nas coordenadas iniciais, orientação e número de movimentos. O predicado *jump* verifica se não existem peças entre as coordenadas iniciais e finais. Por último o predicado *no\_land\_over\_own\_piece* verifica se nas coordenadas finais não está uma peça do próprio jogador.

A implementação dos vários predicados é apresentada a seguir.

```
valid_move(Player, Board, Row, Column, N_moves, Row_valid, Column_valid, N_moves_valid):-
```

```
    inbounds(Row, Column),
    N_moves \= 0,
    piece(Player, Board, Row, Column, Orientation, Type),
    odd(N_moves, Type),
    position(Row, Column, N_moves, Orientation, Row1, Column1),
    nojump(Board, Row, Column, N_moves, Orientation),
    no_land_over_own_piece(Board, Player, Row1, Column1),
    !,
    Row_valid = Row,
    Column_valid = Column,
    N_moves_valid = N_moves.
```

```
valid_move(Player, Board, Row, Column, N_moves, Row_valid, Column_valid, N_moves_valid):-
```

```
    inbounds(Row, Column),
    N_moves \= 0,
    piece(Player, Board, Row, Column, Orientation, Type),
    odd(N_moves, Type),
    position(Row, Column, N_moves, Orientation, _, _),
    nojump(Board, Row, Column, N_moves, Orientation),
    write('Cannot move over one of your pieces. Try again. '), nl, nl,
    !,
    choose_move(Player, human, Board, Row_valid, Column_valid, N_moves_valid).
```

```
valid_move(Player, Board, Row, Column, N_moves, Row_valid, Column_valid, N_moves_valid):-
```

```
    inbounds(Row, Column),
    N_moves \= 0,
    piece(Player, Board, Row, Column, Orientation, Type),
    odd(N_moves, Type),
    position(Row, Column, N_moves, Orientation, _, _),
    write('Cannot jump over other pieces. Try again. '), nl, nl,
    !,
    choose_move(Player, human, Board, Row_valid, Column_valid, N_moves_valid).
```

```
valid_move(Player, Board, Row, Column, N_moves, Row_valid, Column_valid, N_moves_valid):-
```

```
    inbounds(Row, Column),
```

```

N_moves ~= 0,
piece(Player, Board, Row, Column, _, Type),
odd(N_moves, Type),
write('Not a valid number of steps (out of bounds). Try again. '), nl, nl,
!,
choose_move(Player, human, Board, Row_valid, Column_valid, N_moves_valid).
valid_move(Player, Board, Row, Column, N_moves, Row_valid, Column_valid, N_moves_valid):-
    inbounds(Row, Column),
    N_moves ~= 0,
    piece(Player, Board, Row, Column, _, _),
    write('Not a valid number of steps (piece is minion, must be odd). Try again. '), nl, nl,
    !,
    choose_move(Player, human, Board, Row_valid, Column_valid, N_moves_valid).
valid_move(Player, Board, Row, Column, N_moves, Row_valid, Column_valid, N_moves_valid):-
    inbounds(Row, Column),
    N_moves ~= 0,
    write('Not a valid piece. Try again. '), nl, nl,
    !,
    choose_move(Player, human, Board, Row_valid, Column_valid, N_moves_valid).
valid_move(Player, Board, Row, Column, _, Row_valid, Column_valid, N_moves_valid):-
    inbounds(Row, Column),
    write('Number of moves cannot be zero. Try again. '), nl, nl,
    !,
    choose_move(Player, human, Board, Row_valid, Column_valid, N_moves_valid).
valid_move(Player, Board, _, _, _, Row_valid, Column_valid, N_moves_valid):-
    write('Not a valid position in the board. Try again. '), nl, nl,
    !,
    choose_move(Player, human, Board, Row_valid, Column_valid, N_moves_valid).
valid_move(Player, Board, Row, Column, N_moves):-
    column(Column, _),
    row(Row),
    nmoves(N_moves),
    piece(Player, Board, Row, Column, Orientation, Type),
    odd(N_moves, Type),
    position(Row, Column, N_moves, Orientation, Row1, Column1),
    %inbounds(Row1, Column1),
    nojump(Board, Row, Column, N_moves, Orientation),
    no_land_over_own_piece(Board, Player, Row1, Column1).

```

### 3.4 Execução de Jogadas

Os jogadores fazem jogadas à vez enquanto não se verificar nenhuma condição de fim de jogo. No caso de um jogador ter sido eliminado ou não ter nenhuma jogada possível passa a vez. É utilizado o predicado *play* que tem como parâmetros o jogador de quem é a vez de jogar, o número de jogadores, uma lista com informações dos jogadores (tipo de jogador e ordem pela qual os jogadores promoveram o seu primeiro master) e o tabuleiro. O predicado é recursivo passando a vez para o jogador seguinte cada vez que se chama a si mesmo até encontrar uma condição de fim de jogo. Cada jogada corresponde a encontrar um movimento legal (através do *input* do utilizador ou automaticamente no caso do computador) e alterar o tabuleiro de acordo com essa jogada.

A implementação do predicado é apresentada de seguida.

```
play(_, _, Players, Board):-
    game_over(Board, Players, Winner, WinClause),
    !,
    player(Winner, 'M', Letter),
    write('Game Ended'), nl, nl, write('Player '), write(Letter), write(' is the winner (')', write(WinClause), write(')!').

play(Player, N_Players, Players, Board):-
    player_eliminated(Player, Board),
    !,
    player(Player, 'M', Letter),
    write('Player '), write(Letter), write(' has been eliminated. '), nl, nl,
    nextPlayer(Player, N_Players, Player1),
    play(Player1, N_Players, Players, Board).

play(Player, N_Players, Players, Board):-
    no_valid_moves(Player, Board),
    !,
    player(Player, 'M', Letter),
    write('Player '), write(Letter), write(' is skipping is turn because there are no valid moves. '), nl, nl,
    nextPlayer(Player, N_Players, Player1),
    play(Player1, N_Players, Players, Board).

play(Player, N_Players, Players, Board):-
    player(Player, 'M', Letter),
    write('Player '), write(Letter), write(', make your move. '), nl,
    playerType(Player, Players, PlayerType),
    choose_move(Player, PlayerType, Board, Row, Column, N_moves), nl,
    move(Player, Players, Players1, Board, Board1, Row, Column, N_moves),
    display_game(Board1),
    nextPlayer(Player, N_Players, Player1),
    play(Player1, N_Players, Players1, Board1).
```

### 3.5 Avaliação do Tabuleiro e Jogada do Computador

Tendo em conta a complexidade da estratégia e da dificuldade em avaliar de forma quantitativa a melhor jogada as jogadas do computador consistem em procurar todas as jogadas válidas e dentro dessas procurar as jogadas que permitem promover *minions* e as jogadas que permitem eliminar peças do adversário. De seguida escolhe uma jogada aleatória seguindo as seguintes prioridades: se existir uma jogada para promover um *minion* escolhe uma jogada aleatória dessa lista; caso contrário se existir uma jogada para eliminar uma peça do adversário escolhe uma jogada aleatória dessa lista; caso contrário escolhe uma jogada qualquer aleatoriamente.

O predicado *choose\_move* procura todas as jogadas possíveis utilizando os predicados auxiliares *corner\_moves* e *eliminate\_moves* para encontrar as jogadas que permitem promover *minions* e as jogadas que permitem eliminar peças do adversário respetivamente, utilizado depois o predicado *choose\_priority* para obter uma jogada aleatória dessas listas de acordo com as prioridades estabelecidas.

```
choose_move(Player, computer, Board, Row, Column, N_moves):-
```

```
    findall([Row, Column, N_moves], valid_move(Player, Board, Row, Column, N_moves), List_moves),
    corner_moves(Board, List_moves, List_corners),
    eliminate_moves(Board, List_moves, List_eliminate),
    choose_priority(List_moves, List_eliminate, List_corners, Row, Column, N_moves),
    sleep(1),
    write('[Computer] Row: '), write(Row), write(', Column: '), write(Column), write(', Number of moves: '), write(N_moves), nl,
    sleep(1).
```

```
choose_priority(List, [], [], Row, Column, N_moves):-
```

```
    length(List, Size),
    Max is Size + 1,
    random(1, Max, Rand),
    nth1(Rand, List, [Row, Column, N_moves]).
```

```
choose_priority(_, List, [], Row, Column, N_moves):-
```

```
    length(List, Size),
    Max is Size + 1,
    random(1, Max, Rand),
    nth1(Rand, List, [Row, Column, N_moves]).
```

```
choose_priority(_, _, List, Row, Column, N_moves):-
```

```
    length(List, Size),
    Max is Size + 1,
    random(1, Max, Rand),
    nth1(Rand, List, [Row, Column, N_moves]).
```

```
corner_moves(_, [], []).
```

```
corner_moves(Board, [[Row, Column, N_moves]|T], [[Row, Column, N_moves]|T1]):-
```

```
    piece(_, Board, Row, Column, Orientation, m),
    position(Row, Column, N_moves, Orientation, Row1, Column1),
    corner(Row1, Column1),
    !,
    corner_moves(Board, T, T1).
```

```
corner_moves(Board, [_|T], T1):- corner_moves(Board, T, T1).
```

```

eliminate_moves(_, [], []).
eliminate_moves(Board, [[Row, Column, N_moves]|T], [[Row, Column, N_moves]|T1]):-
    piece(_, Board, Row, Column, Orientation, _),
    position(Row, Column, N_moves, Orientation, Row1, Column1),
    piece(_, Board, Row1, Column1, _, _),
    !,
    eliminate_moves(Board, T, T1).
eliminate_moves(Board, [_|T], T1):- eliminate_moves(Board, T, T1).

```



### 3.6 Final do Jogo

Para verificar se se observa alguma das duas condições de fim de jogo é utilizado o predicado *game\_over* que utiliza os predicados *no\_minions* e *only\_one\_player* que verificam respetivamente se já não existem *minions* em jogo e/ou se já só existe um jogador em jogo. No caso de só existir um jogador está encontrado o vencedor. No caso de só existirem masters é utilizado o predicado *mostMasters* para obter o vencedor. O predicado *mostMasters* utiliza o predicado auxiliar *count\_masters* para obter o número de masters de cada jogador e o predicado auxiliar *check\_tie* para verificar no caso de empate quem promoveu primeiro uma peça utilizando para isso o predicado *firstMaster*.

```
game_over(Board, Players, Winner, WinClause):- no_minions(Board, Players, Winner, WinClause).
```

```
game_over(Board, _, Winner, 'Last player standing'):- only_one_player(Board, Winner).
```

```
no_minions(Board, _, _, _):-
```

```
    member(Line, Board),
```

```
    member(Square, Line),
```

```
    member(m, Square),
```

```
    !,
```

```
    fail.
```

```
no_minions(Board, Players, Winner, WinClause):-
```

```
    mostMasters(Board, Players, Winner, WinClause).
```

```
mostMasters(Board, Players, Winner, WinClause):-
```

```
    count_masters(1, Board, Players, [], List),
```

```
    check_tie(List, Players, Winner, WinClause).
```

```
count_masters(Player, _, Players, List, List):-
```

```
    length(Players,N),
```

```
    Player > N.
```

```
count_masters(Player, Board, Players, List, List2):-
```

```
    count_player_masters(Player, 0, Board, List, List1),
```

```
    Player1 is Player + 1,
```

```
    count_masters(Player1, Board, Players, List1, List2).
```

```
count_player_masters(_, Sum, [], List, List1):-
```

```
    append(List, [Sum], List1).
```

```
count_player_masters(Player, Sum, [H|T], List, List1):-
```

```
    delete(H, [Player, _, _], Rest),
```

```
    delete(Rest, [Player, _, _], Rest1),
```

```
    delete(Rest1, [Player, _, _], Rest2),
```

```
    delete(Rest2, [Player, _, _], Rest3),
```

```
    length(Rest3,N),
```

```
    Sum1 is Sum + 8 - N,
```

```
    count_player_masters(Player, Sum1, T, List, List1).
```

```
check_tie(List, Players, Winner, 'First Master'):-
```

```

        max_member(Max, List),
        nth1(P1, List, Max),
        nth1(P2, List, Max),
        P1 \= P2,
        !,
        firstMaster(List, Max, 1, Players, Winner).
check_tie(List, _, Winner, 'Most Minions'):-
    max_member(Max, List),
    nth1(Winner, List, Max).

firstMaster(List, Max, Order, Players, Winner):-
    nth1(Winner, Players, [_, Order]),
    nth1(Winner, List, Max).
firstMaster(List, Max, Order, Players, Winner):-
    Order1 is Order + 1,
    firstMaster(List, Max, Order1, Players, Winner).

only_one_player(Board, _):-
    member(Line1, Board),
    member([P1,_,_], Line1),
    member(Line2, Board),
    member([P2,_,_], Line2),
    P1 \= P2,
    !,
    fail.
only_one_player(Board, Player):-
    member(Line1, Board),
    member([Player,_,_], Line1).

```

## 4 Interface com o Utilizador

Para correr o jogo, o utilizador deve correr o comando ‘pivot.’, que lança o jogo. De seguida, o utilizador é saudado e é-lhe pedida a informação em relação ao número de jogadores (2 a 4), e quantos dos quais são humanos ou ‘computador’, sendo possíveis todas as combinações. (Figura 5.1)



```
SICStus 4.2.3 (x86_64-win32-nt-4): Sun Oct 7 18:55:53 WEDT 2012
File Edit Flags Settings Help
sec 86752 bytes
| 7- pivot.
% loading c:/program files/sicstus prolog vc10 4.2.3/library/lists.po...
% module lists imported into user
% loading c:/program files/sicstus prolog vc10 4.2.3/library/types.po...
% module types imported into lists
% loaded c:/program files/sicstus prolog vc10 4.2.3/library/types.po in module
types. 0 msec 1440 bytes
% loaded c:/program files/sicstus prolog vc10 4.2.3/library/lists.po in module
lists. 0 msec 125264 bytes
% loading c:/program files/sicstus prolog vc10 4.2.3/library/random.po...
% module random imported into user
% module types imported into random
% loading foreign resource c:/program files/sicstus prolog vc10 4.2.3/library/
x86_64-win32-nt-4/random.dll in module random
% loaded c:/program files/sicstus prolog vc10 4.2.3/library/random.po in module
random. 15 msec 24528 bytes
% loading c:/program files/sicstus prolog vc10 4.2.3/library/system.po...
% module system imported into user
% module types imported into system
% loaded c:/program files/sicstus prolog vc10 4.2.3/library/system.po in module
system. 0 msec 5280 bytes

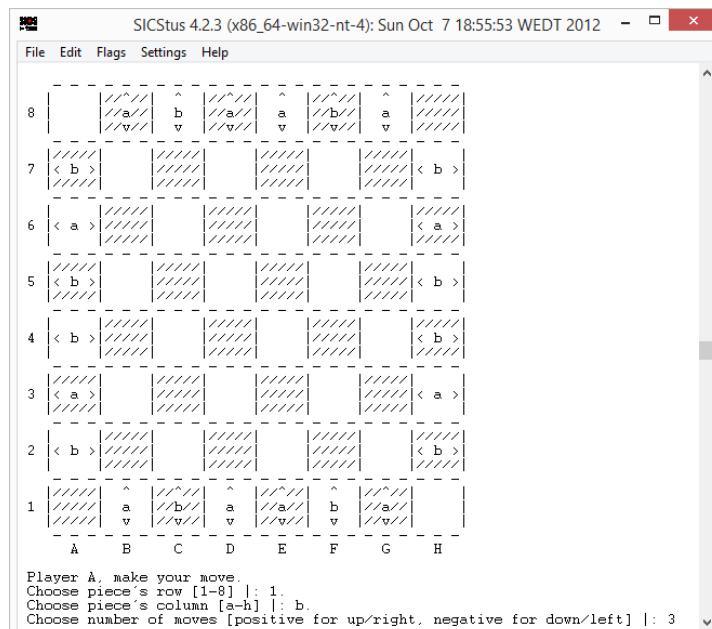
-----Welcome to Pivit!-----

Enter number of players [2-4]: 2.

Choose human and computer players.
Player 1?[human or pc]: human.
Player 2?[human or pc]: PC.
```

Figura 5.1 – Iniciar o Jogo

De seguida é mostrado o tabuleiro, e solicita-se a jogada do primeiro jogador. No caso de ser um jogador humano, este indica a linha e a coluna da peça, e o número de movimentos (número de ‘casas’ a andar). Os números das linhas e das colunas: de 1 a 8 de baixo para cima, para permitir um formato mais intuitivo e semelhante ao xadrez (ver 4.2). (Figura 5.2)



```
SICStus 4.2.3 (x86_64-win32-nt-4): Sun Oct 7 18:55:53 WEDT 2012
File Edit Flags Settings Help

  8 |   |   | ^ |   | ^ |   | ^ |   |
    |   | a | b |   | a | b |   |
    |   | v | v |   | v | v |   |
  7 | < b > |   |   |   |   |   |   | < b > |
    |   |   |   |   |   |   |   |   |
  6 | < a > |   |   |   |   |   |   | < a > |
    |   |   |   |   |   |   |   |   |
  5 | < b > |   |   |   |   |   |   | < b > |
    |   |   |   |   |   |   |   |   |
  4 | < b > |   |   |   |   |   |   | < b > |
    |   |   |   |   |   |   |   |   |
  3 | < a > |   |   |   |   |   |   | < a > |
    |   |   |   |   |   |   |   |   |
  2 | < b > |   |   |   |   |   |   | < b > |
    |   |   |   |   |   |   |   |   |
  1 |   | a | b | a | a | b | a |   |
    |   | v | v | v | v | v | v |   |
    A   B   C   D   E   F   G   H

Player A, make your move.
Choose piece's row [1-8] |: 1.
Choose piece's column [a-h] |: b.
Choose number of moves [positive for up/right, negative for down/left] |: 3
```

Figura 5.2 – Fazer um Movimento

O tabuleiro atualizado é então mostrado na consola e solicita-se o movimento dos jogadores seguintes até ao fim do jogo. Para manter a interface uniforme, também se mostra o tabuleiro antes do ‘computador’ fazer a sua jogada, assim como um *prompt* para tal. (Figura 5.3) Momentos depois, a jogada é realizada e passa a vez ao jogador seguinte. (Figura 5.4)

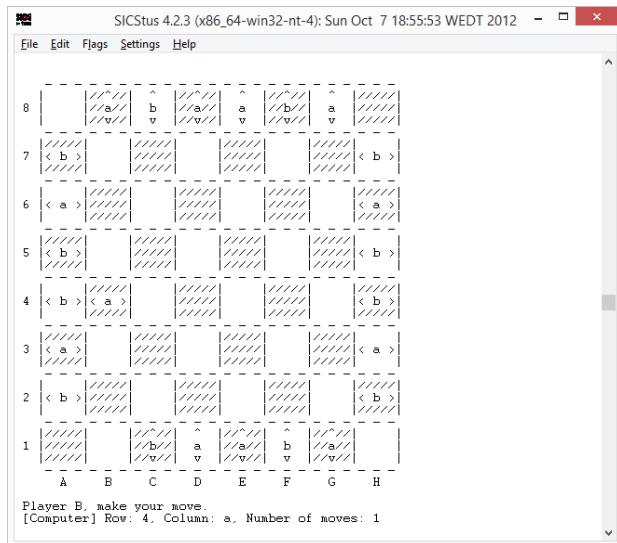


Figura 5.3 – Movimento do computador

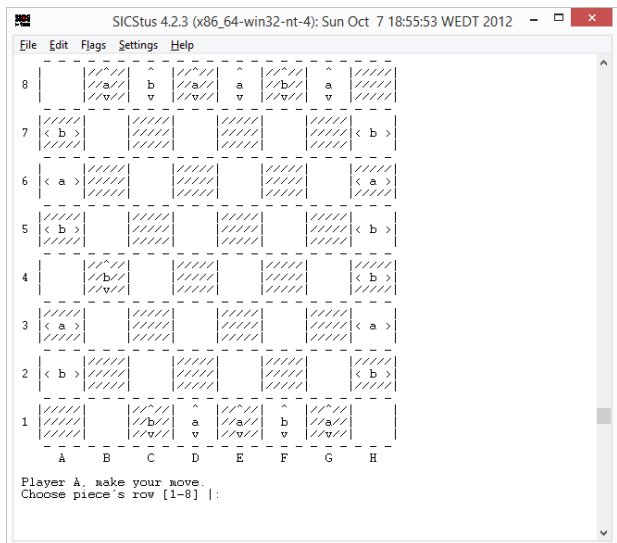


Figura 5.4 – Atualização do tabuleiro e jogada seguinte

## 5 Conclusões

Este trabalho, para além da familiarização com a linguagem Prolog, permitiu perceber as potencialidades da programação em lógica, nomeadamente do *backtracking*, e como este paradigma se adequa melhor a problemas de satisfação de restrições e de otimização como é o caso de um jogo de tabuleiro.

Seria possível otimizar a “inteligência artificial” se houvesse mais experiência com a estratégia do jogo e disponibilidade de tempo mas considerando o propósito deste trabalho prático a implementação do jogo apresentada parece-nos cumprir os objetivos pretendidos.

## Bibliografia

Os seguintes websites foram utilizados para obter informação e recursos sobre o Jogo de Tabuleiro:

Website oficial do jogo: <http://pivitgame.com/>

Pivit no BoardGameGeek: <http://www.boardgamegeek.com/boardgame/135473/pivit>

Pivit no Kickstarter: <https://www.kickstarter.com/projects/913572758/pivit>

Video promocional do criador do jogo: <https://www.youtube.com/watch?v=yBAoF01kDV4>

## Anexos

O código é incluído no arquivo .zip submetido no moodle com o relatório.