Xero-G

Relatório Final

Mestrado Integrado em Engenharia Informática e Computação

Programação em Lógica

Grupo Xero-G_1
Diogo Rafael Amorim Mendes - up201605360
Simão Pereira de Oliveira - up201603173

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

17 de Novembro de 2019

Índice

Resumo

Xero-G é um jogo disputado entre 2 jogadores, tratando-se de um jogo de estratégia sem nenhum elemento de sorte. Neste trabalho foi realizada a implementação deste jogo na linguagem *Prolog.*

O jogo tem por base um tabuleiro quadriculado de 6 linhas por 6 colunas, ao qual se acrescenta mais 1 linha em cada extremidade que representam os objetivos. Assim, cada jogador tem uma base, representada pela linha mais próxima de si, e o seu objetivo é chegar à base do adversário. As peças têm valores diferentes que indicam quantas casas podem andar, sendo que o movimento é ortogonal.

Introdução

No âmbito da unidade curricular de Programação em Lógica do 3º ano do Mestrado Integrado em Engenharia Informática e de Computação, foi-nos sugerido o desenvolvimento de um jogo utilizando a linguagem *Prolog*. De entre as opções que nos foram disponibilizadas, foi escolhido o jogo *Xero-G*.

O objetivo deste trabalho foi o contacto com a linguagem *Prolog* que até à realização deste trabalho nos era desconhecida, e a aplicação de conceitos referidos nas aulas teóricas.

O relatório está estruturado da seguinte forma:

- **O Jogo Xero-G:** Descrição sucinta do jogo, história e regras.
- **Lógica do Jogo:** Descrição do projeto e da sua implementação em Prolog, estruturado da seguinte forma:
    - Representação do Estado do Jogo: Estados iniciais, intermédios e finais do jogo.
    - Visualização do Tabuleiro: Descrição do predicado de visualização.
    - Lista de Jogadas Válidas: Explicação dos predicados utilizados para validar as jogadas.
    - Execução de Jogadas: Ciclo do jogo, validação e execução de cada jogada.
    - Final do Jogo: Verificação do fim do jogo, com identificação do vencedor.
    - Avaliação do Tabuleiro: Explicitação do conteúdo do tabuleiro.
    - Jogada do Computador: Descrição dos predicados responsáveis pela geração de movimentos do computador.
- **Conclusões**
- **Bibliografia**

O Jogo *Xero-G*

Xero-G é inspirado num jogo de estratégia abstracto de 1985, *Gyges*, de Claude Leroy. A única diferença notável entre os dois jogos é o tabuleiro, visto que as regras são relativamente similares.

Esta versão foi criada por Kat Costa e pelo designer Kai Kalhh.

Uma característica interessante deste jogo é o facto de nenhum jogador ser “dono” de uma peça, podendo esta vir a ser utilizada por ambos os jogadores.

Inicialmente, cada jogador coloca as suas naves na linha mais próximo de si, no final devem existir em cada linha duas naves com valor 1, duas com valor 2 e duas com valor 3. Em cada turno, cada jogador é obrigado a jogar com uma peça da linha mais próxima de si. Cada nave pode deslocar se apenas ortogonalmente e o número de vezes correspondente ao seu valor. Durante o movimento da peça, esta só pode passar por quadrados vazios, contudo o seu movimento final pode terminar numa casa ocupada, podendo depois escolher entre duas acções especiais: *Reprogram Coordinates*- o jogador que atualmente controla a nave pode mover a que inicialmente ocupada esse espaço para outro lugar vazio e dentro das 6 linhas; *Rocket Boost*- o jogador pode mover a peça o número de vezes correspondente ao valor da nave que ocupa o espaço. É possível encadear *Rocket Boosts* sendo que um turno termina quando uma nave termina num espaço vazio, ou é utilizado um *Reprogram Coordinates*.

O vencedor é quem conseguir atingir a linha mais próxima do adversário, não a inicial mas a anterior a esta.

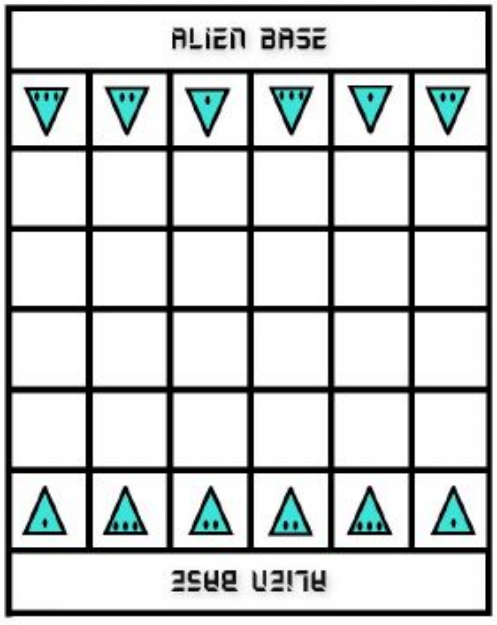

Figura 1: Possível situação inicial de um jogo *Xero-G*

Lógica do Jogo

Representação do Estado do Jogo

O tabuleiro de jogo é representado por uma lista de listas. O tabuleiro é de 8 linhas por 6 colunas, assim temos 8 listas de 6 elementos cada.

Como exemplo, o código abaixo representa em *Prolog*: o tabuleiro vazio, após os 2 jogadores terem colocado as naves, a meio de um jogo e no fim de um. Cada número utilizado na representação interna ao programa é traduzido quando o tabuleiro é representado na consola.

```
/*  board_element(?PieceNumber, ?PieceView)

    Given a PieceNumber unifies PeaceView with the console board repre
sentation of that piece
*/
board_element(0, '     ').
board_element(1, '  1  ').
board_element(2, '  2  ').
board_element(3, '  3  ').
board_element(5, '  0  ').
```

Figura 2: Tradução da representação interna das peças

```
/*  initial_board(+Board)

    Unifies Board with the initial empty game board 6x6 playable area
    Top and Bottom line are the objectives
*/
initial_board([[0,0,0,0,0,0],
               [0,0,0,0,0,0],
               [0,0,0,0,0,0],
               [0,0,0,0,0,0],
               [0,0,0,0,0,0],
               [0,0,0,0,0,0],
               [0,0,0,0,0,0],
               [0,0,0,0,0,0]]).
```

Figura 3: Tabuleiro vazio

```
% Board after ships are placed by both players
ships_placed_board([[0,0,0,0,0,0],
                    [1,2,3,3,2,1],
                    [0,0,0,0,0,0],
                    [0,0,0,0,0,0],
                    [0,0,0,0,0,0],
                    [0,0,0,0,0,0],
                    [1,2,3,1,2,3],
                    [0,0,0,0,0,0]]).
```

Figura 4: Tabuleiro após os jogadores terem colocados as naves

```
% Intermediate board
intermediate_board([[0,0,0,0,0,0],
                    [0,0,0,0,0,0],
                    [1,0,0,0,0,1],
                    [2,0,3,3,2,3],
                    [0,2,0,3,2,0],
                    [1,0,0,0,1,0],
                    [0,0,0,0,0,0],
                    [0,0,0,0,0,0]]).
```

Figura 5: Tabuleiro a meio de um jogo

```
% Final board
final_board([[0,1,0,0,0,0],
             [0,0,0,0,0,0],
             [0,2,0,0,0,1],
             [1,0,3,3,2,3],
             [0,2,0,3,2,0],
             [0,0,0,0,1,0],
             [0,0,0,0,0,0],
             [0,0,0,0,0,0]]).
```

Figura 6: Tabuleiro no fim de um jogo

Visualização do Tabuleiro

As representações internas expostas anteriormente são apresentadas na consola da seguinte forma:

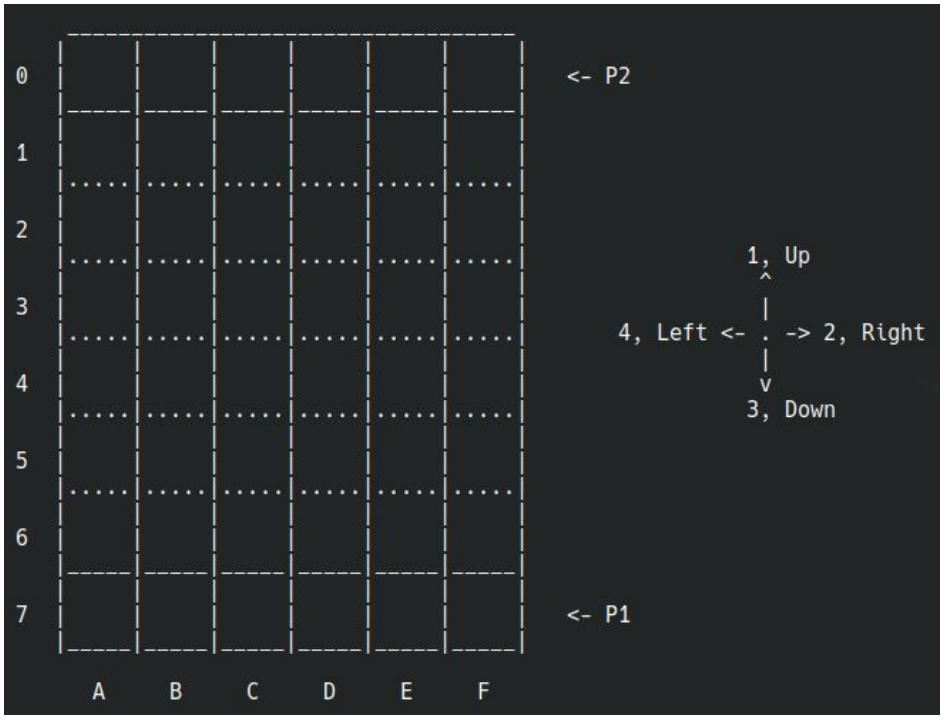

Figura 7: Tabuleiro vazio visto na consola

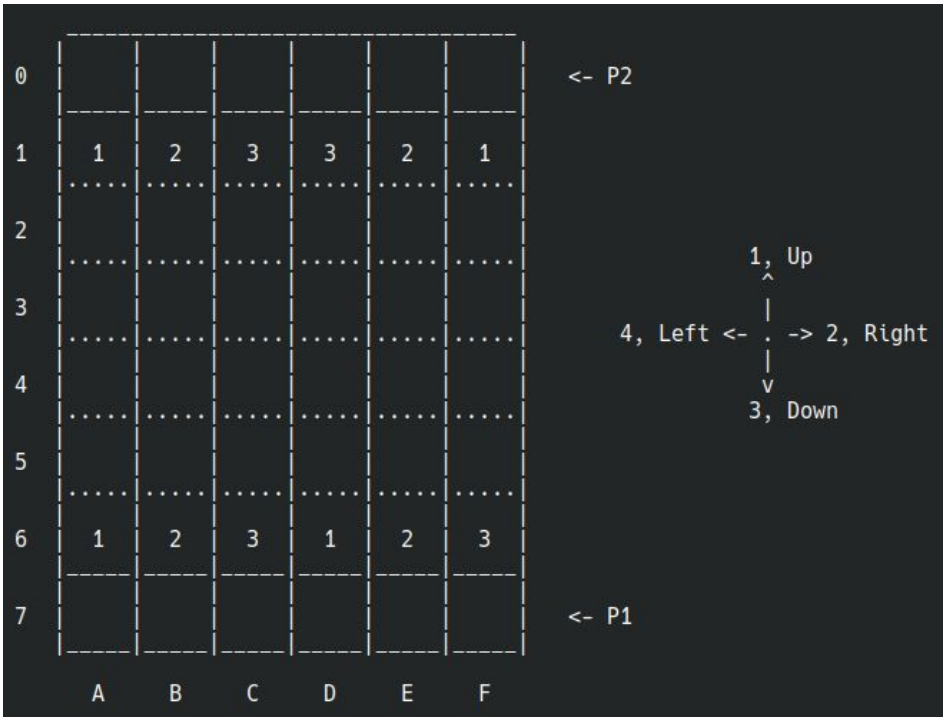

Figura 8: Tabuleiro após os jogadores colocarem as peças, visto na consola

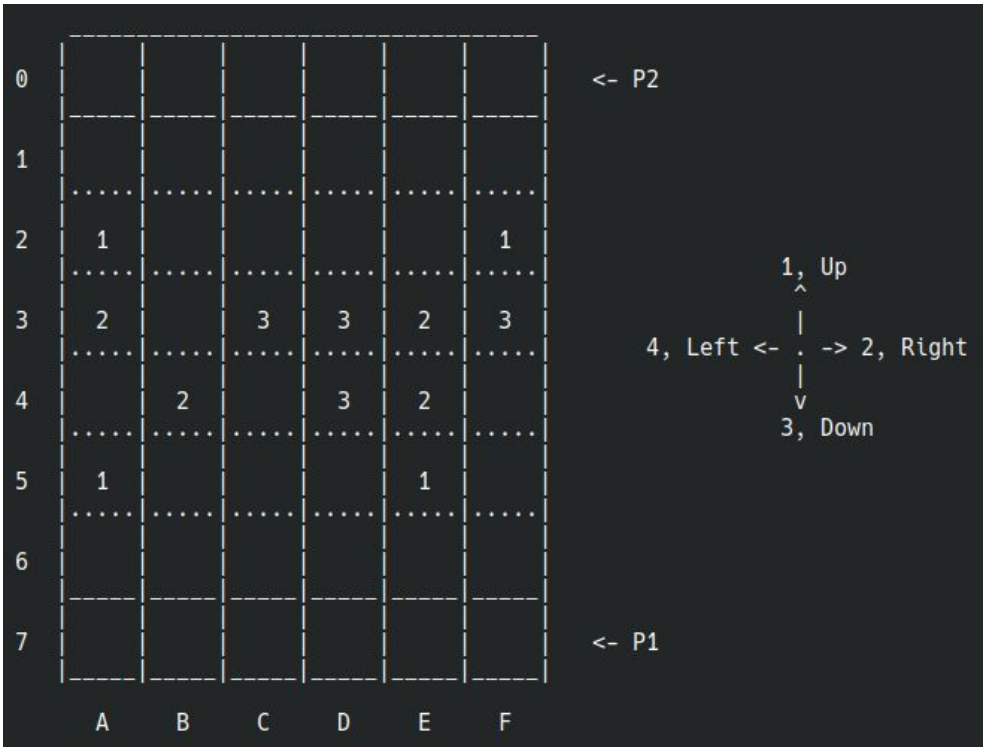

Figura 9: Tabuleiro a meio de um jogo visto na consola

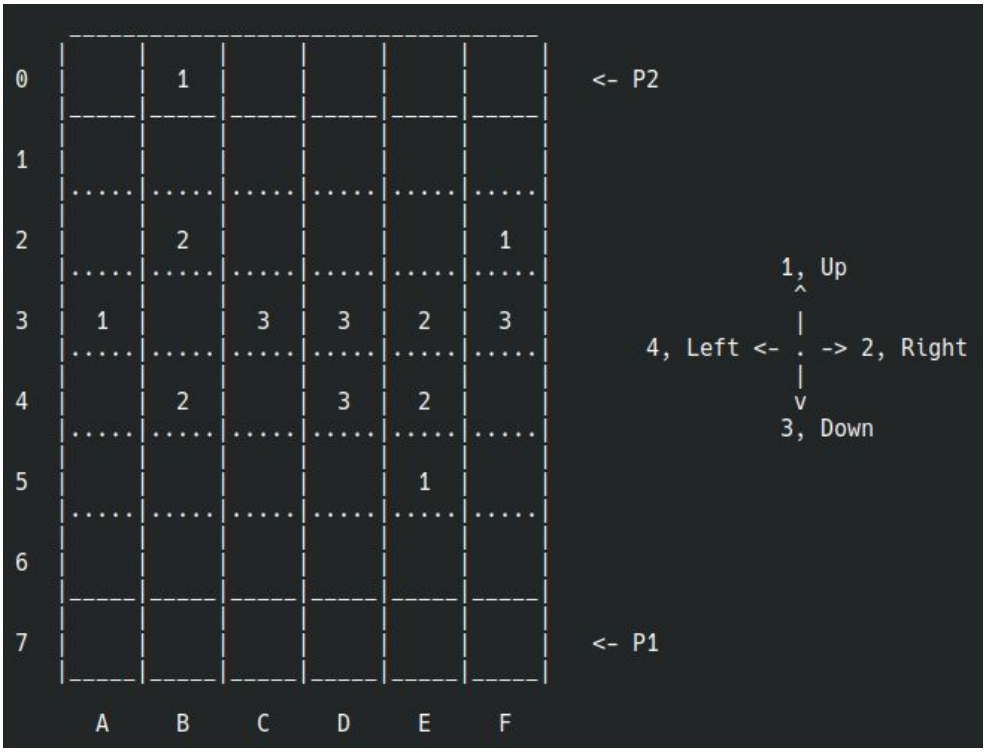

Figura 10: Tabuleiro no fim de um jogo visto na consola

De forma a facilitar a interação do utilizador com o tabuleiro, as linhas foram numeradas, de forma crescente de cima para baixo, e a cada coluna foi associada uma letra, também de forma crescente da direita para a esquerda. Junto do tabuleiro existe também uma bússola que indica as 4 direções possíveis de movimentar uma peça.

Para imprimir o tabuleiro na consola foi construído o predicado *print_board(+Board)* que recebe uma lista de listas, Board, e a imprime na consola. São utilizadas chamadas recursivas ao predicado *print_board_rec(+Board, +LineNumb)* que por sua vez chama o predicado *print_line(+Line)*, também de forma recursiva.

Lista de Jogadas Válidas

As jogadas válidas num tabuleiro dependem do lado do jogador, assim ao predicado sugerido no enunciado do trabalho prático foi necessário adicionar essa condição.

Então, para obter uma listagem das jogadas válidas num tabuleiro faz-se uma chamada ao predicado *valid_moves(+Board, -ValidMoves, +Player)*. Este predicado primeiramente verifica se a linha escolhida foi a correta, isto é, se não existe nenhuma linha entre a escolhida e *Player* com naves. De seguida, simula o movimento dessa peça em 3 direções: direita, esquerda, cima (no caso de p1) ou baixo (no caso de p2).

No fim são colocadas na lista *ValidMoves* as combinações de linhas, coluna e direções que correspondem a jogadas válidas para o jogador no tabuleiro *Board.*

É importante referir que não se verifica a possibilidade de realizar uma acção especial, sendo assim a última posição da nave terá que estar vazia.

```
/*  valid_moves(+Board, -ValidMoves, +Player)

    Returns a list, ValidMoves of the valid moves in Board taking into account which
Player is playing
*/
valid_moves(Board, ValidMoves, Player):-
    findall(Line-Col-Dir, valid_move(Board, Player, Line, Col, Dir), ValidMoves).
```

Figura 11: Predicado que devolve uma lista de jogadas válidas

```
/*  validate_correct_line(+Player, +Board, +Line)

    Given the Player it either checks backwards or forwards if there is any line
    that still has a ship
*/
validate_correct_line(p1, _, 7):-!.
validate_correct_line(p1, Board, LineNum) :-
    BefLine is LineNum +1,
    nth0(BefLine, Board, Line),
    empty(Line),
    validate_correct_line(p1, Board, BefLine).
```

Figura 11: Predicado que valida se a linha escolhida por p1 é a correta

Execução de Jogadas

Todos os os jogos, com ou sem bot, ocorrem dentro do predicado *play_game([...])* que é chamado recursivamente dependendo do modo de jogo. Foram construídas 8 versões do predicado:

- Predicados utilizados no modo 2 jogadores:
    - *play_game([+Board, +Player])*: Inicialmente *Board* é igual ao tabuleiro vazio, por isso neste predicado cada jogador coloca as naves nas respectivas linhas
    - *play_game([+Board, +Player]):* Predicado de uma jogada normal entre 2 jogadores
- Predicados utilizados no modo jogador versus computador:
    - *play_game([+Board, +BotDiff, +FirstPlayer]):* Inicialmente *Board* é igual ao tabuleiro vazio, então é pedido ao primeiro jogador que preencha a sua linha e de seguida ao outro jogador. Neste caso um dos jogadores pode ser um *bot*
    - *play_game([+Board, +BotDiff, +FirstPlayer]):* Predicado de uma jogada normal, existem variações deste predicado dependendo de quem for o primeiro jogador e de forma a respeitar a ordem
- Predicados utilizados no modo computador versus computador:
    - *play_game([+Board, +BotDiff, +BotDiff]):* Inicialmente *Board* está vazio então cada *bot* preenche a sua linha com as suas naves
    - *play_game([+Board, +BotDiff, +BotDiff]):* Predicado de um jogada normal em que cada *bot* joga na sua vez

Dentro do predicado *play_game*, no modo de jogo 2 jogadores, são realizadas chamadas recursivas ao predicado *player_turn(+Board, -NewBoard, +Player, -NextPlayer),* que imprime o tabuleiro atual e chama o predicado *move(+Player, +Board, -NewBoard).*

Este predicado pede ao jogador as coordenadas da nave que pretende mover, *get_coord(-Line, -Col)*, de seguida é feita a verificação se não existem outras naves em linhas anteriores, *validade_correct_line(+Player, +Board, +Line)*, e se às coordenadas escolhidas corresponde uma nave, isto é, se o jogador não escolheu um lugar vazio.

Após esta confirmação é chamado o predicado *player_move(+Ship, +NMoves, +Line, +Col, +Board, -NewBoard)*. No caso em que *NMoves* é diferente de 1, o jogador só pode escolher direções em que a nave passe a ocupar um espaço vazio. Por outro lado, quando *NMoves* é 1 a nave tanto pode ocupar um espaço vazio, que termina a jogada, ou ocupar um espaço ocupado que dá acesso a uma ação especial.

```
/*  player_move(+Ship, +NMoves, +Line, +Col, +Board, -NewBoard)

    Asks for the direction to move Ship from Line Col
    If its a middle move the cell must be empty
    If its the last move also asks for power move
    In the end unifies Ship new location with NewBoard
    The last move can trigger a power move if Ship falls on an occupied square
*/
player_move(Ship, 1, Line, Col, Board, NewBoard):-
    repeat,
    get_direction(Dir),
    get_next_coords(Dir, Line, Col, NewLine, NewCol, Flag),
    Flag =\= 1,
    get_piece(NewLine, NewCol, Board, Piece),
    set_cell(Line, Col, 0, Board, NewBoard1),
    (Piece = 0 ->
        set_cell(NewLine, NewCol, Ship, NewBoard1, NewBoard)
        ;
        choose_power_move(Move),
        (Move = 1 ->
            reprogram_coordinates(Piece, Ship, NewLine, NewCol, NewBoard1, NewBoard)
            ;
            rocket_boost(Piece, Ship, NewLine, NewCol, NewBoard1, NewBoard)
        )
    ),
    !.
```

Figura 12: Predicado *player_move* quando *NMoves* é 1

A escolha entre as 2 ações especiais, explicadas nas regras do jogo, é feita pelo predicado *choose_powe_move(-Move)*, onde o utilizador decide ativar a função *reprogram_coordinates(+Piece, +Ship, +Line, +Col, +Board, -NewBoard)* ou *rocket_boost(+NMoves, +Ship, +Line, +Col, +Board, -NewBoard)*.

A primeira espera pela introdução das novas coordenadas da peça por parte do utilizador, verificando se o lugar escolhido está vazio e se as coordenadas são válidas. Na segunda opção o jogador essencialmente volta a poder jogar mais um turno, logo é feita a chamada ao predicado *player_move(+Ship, +NMoves, +Line, +Col, +Board, -NewBoard)*.

No final da chamada a *player_turn* é feita uma chamada ao predicado *game_over(+Player, +Board)* que verifica se o jogador atual ganhou o jogo.

Final do Jogo

Tal como referido em Execução de Jogadas no fim de cada turno de um jogador é feita a chamada ao predicado *game_over(+Player, +Board)*.

Este predicado verifica se a base do jogador adversário está vazia ou não, isto é, no caso de *Player* corresponder a p1 é feita a verificação da linha 0, no caso de *Player* corresponder a p2 é feita da linha 7. Se a linha estiver ocupada significa que *Player* ganhou o jogo, caso contrário a chamada simplesmente falha procedendo à jogada seguinte.

```
/*  game_over(+Player, +Board)

    Given Player cheks if line 0 or 7 aren't empty if so it means Player won
*/
game_over(p1, Board):-
    nth0(0, Board, Line),
    \+ empty(Line),
    clear_console,
    print_board(Board),
    print_player_won(p1),
    get_any_key,
    main_menu, !.

% Calls to other predicats failed, means neither p1 or p2 have won
game_over(_, _).
```

Figura 13: Predicado *game_over* para p1. Segundo predicado no caso de ninguém vencer

Avaliação do Tabuleiro

O predicado *value_move(+Board, +Player, +Line, +Col, +Dir, -Val)* atribui um pontuação a uma jogada com base na proximidade à base do adversário. Este predicado simula o movimento da peça em (*Line*, *Col*) na direção *Dir* e de seguida chama o predicado *value(+Player, +Board, -Val)*. Como a pontuação depende do jogador foram concebidos duas versões deste predicado, de forma a tornar o código mais legível.

Considerando a seguinte matriz como tabuleiro do jogo:

$$\begin{bmatrix} x_{01} & x_{02} & x_{03} & x_{04} & x_{05} & x_{06} \\ x_{11} & x_{12} & x_{13} & x_{14} & x_{15} & x_{16} \\ x_{21} & x_{22} & x_{23} & x_{24} & x_{25} & x_{26} \\ x_{31} & x_{32} & x_{33} & x_{34} & x_{35} & x_{36} \\ x_{41} & x_{42} & x_{43} & x_{44} & x_{45} & x_{46} \\ x_{51} & x_{52} & x_{53} & x_{54} & x_{55} & x_{56} \\ x_{61} & x_{62} & x_{63} & x_{64} & x_{65} & x_{66} \end{bmatrix}$$

O valor das pontuações de uma jogada realizada por p1 (val_1) é dado por:

$$val_1 = 20\sum_{i=1}^{6} x_{0i} + 12\sum_{i=1}^{6} x_{1i} + 10\sum_{i=1}^{6} x_{2i} + 8\sum_{i=1}^{6} x_{3i} + 6\sum_{i=1}^{6} x_{4i} + 5\sum_{i=1}^{6} x_{5i} + \sum_{i=1}^{6} x_{6i}$$

O valor das pontuações de uma jogada realizada por p2 (val_2) é dado por:

$$val_2 = \sum_{i=1}^{6} x_{1i} + 5\sum_{i=1}^{6} x_{2i} + 6\sum_{i=1}^{6} x_{3i} + 8\sum_{i=1}^{6} x_{4i} + 10\sum_{i=1}^{6} x_{5i} + 12\sum_{i=1}^{6} x_{6i} + 20\sum_{i=1}^{6} x_{7i}$$

Jogada do Computador

A possibilidade de jogar contra o computador ou de assistir a um jogo completamente automatizado é apresentada ao utilizador no menu inicial do jogo.

Caso o jogador decida jogar contra um *bot* o predicado `play_game` toma a seguinte forma: *play_game([+Board, +BotDiff, +FirstPlayer])*.

No início do jogo, o computador precisa de posicionar as suas peças na fila inicial usando o predicado *bot_set_ships(+Board, +Row, -NewBoard)* que de forma aleatória as coloca no tabuleiro.

A escolha do movimento do *bot* é feita usando o predicado *choose_move(+BotDiff, +Board, -NewBoard, +Player)* . Uma vez que existem 2 dificuldades denominadas *Random* e *'Smart'* foram criadas duas versões deste predicado.

Na primeira dificuldade o computador através do predicado *valid_moves*, apresentado anteriormente, obtém uma lista de jogadas possíveis para o tabuleiro atual. Dessas jogadas possíveis o *bot* vai escolher uma de forma aleatória e proceder à sua execução.

Na dificuldade *'Smart'* o *bot* obtém uma lista de jogadas válidas como na dificuldade *random*, mas a cada possível jogada está associado um valor. Esse valor, calculado pelo predicado *value*, que indica qual a melhor jogada, pelos parâmetros já apresentados. Neste caso, é escolhida a jogada com maior valor.

Após escolhida jogada é feita a chamada ao predicado *bot_move(+Line, +Col, +Board, -NewBoard)* que realiza o movimento da peça em (*Line*, *Col*) na direção *Dir* e unifica o resultado em *NewBoard*.

Como já foi referido em Lista de Jogadas Válidas, o predicado *valid_moves* apenas tem em consideração movimentos básicos ignorando as duas ações especiais possíveis, por isso, a lista de jogadas possíveis dos *bots* rapidamente diminui. Por causa desss situações foi criado um predicado *bot_cant_move(+BotDiff, +Board)* que dá a vitória ao jogador.

Outra situação que nos deparamos por causa desta limitação foi no modo computador contra computador, neste cenário o jogo facilmente entra numa situação de ciclo infinito. Após realizar todas as jogadas para a frente, restam apenas uma ou duas peças na linha anterior cujas jogadas válidas são para a direita ou para a esquerda. Então ambos os *bots* movem essas peças lateralmente e o jogo não altera o estado. Esta situação no modo jogador contra computador pode ocorrer, mas como se trata de um jogador humano é possível alcançar o fim do jogo.

```
/* choose_move(+BotDiff, +Board, -NewBoard, +Player)

    Depending on BotDiff calculates Line and Col of next move, unifies with NewBoard
*/
choose_move(random, Board, NewBoard, Player):-
    valid_moves(Board, ValidMoves, Player),
    length(ValidMoves, NMoves),
    (NMoves = 0 ->
        bot_cant_move(random, Board)
        ;
        random(0, NMoves, Move),
        nth0(Move, ValidMoves, Line-Col-Dir),
        print_bot_move(Line, Col, Dir),
        bot_move(Line, Col, Dir, Board, NewBoard)
    ).
```

Figura 14: Predicado *choose_move* na dificuldade *random*

Conclusões

Com a realização deste trabalho o grupo teve um contacto mais prático com a linguagem *Prolog* em comparação com a abordagem mais teórica das aulas. Desta forma foi-nos possível aprender mais sobre o paradigma de programação nesta linguagem e os fundamentos de Programação em Lógica.

Durante o desenvolvimento deste projecto encontramos algumas dificuldades, a principal foram as diferenças acentuadas entre a linguagem *Prolog* e as linguagens utilizadas no curso até agora.

Depois de ultrapassar este obstáculo, outro desafio encontrado foi a recursividade que numa parte inicial não estava a ser utilizada corretamente. Contudo, foi possível arranjar soluções para todos os problemas encontrados, umas com mais perícia do que outras, e ao mesmo tempo manter uma boa organização do código.

Existem então, aspectos que poderiam ser melhorados no nosso código como o cálculo de jogadas válidas que, principalmente por falta de tempo, poderia ter em conta as duas acções especiais e ainda a eficiência de certas soluções utilizadas.

Finalmente, o projeto foi concluído com sucesso e o seu desenvolvimento foi chave para uma melhor compreensão da linguagem *Prolog*.

Bibliografia

https://www.boardgamegeek.com/boardgame/282976/xero-g
https://sicstus.sics.se/sicstus/docs/3.7.1/html/sicstus_44.html
https://sicstus.sics.se/sicstus/docs/latest/html/sicstus/lib_002drandom.html#lib_002drandom
https://sicstus.sics.se/sicstus/docs/latest/html/sicstus/lib_002dlists.html#lib_002dlists

Anexo I - Interface com o utilizador

Figura 15: Menu inicial

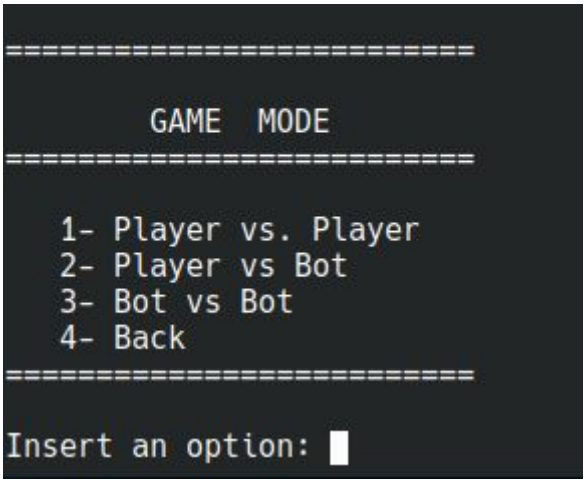

Figura 16: Menu de escolha do modo de jogo

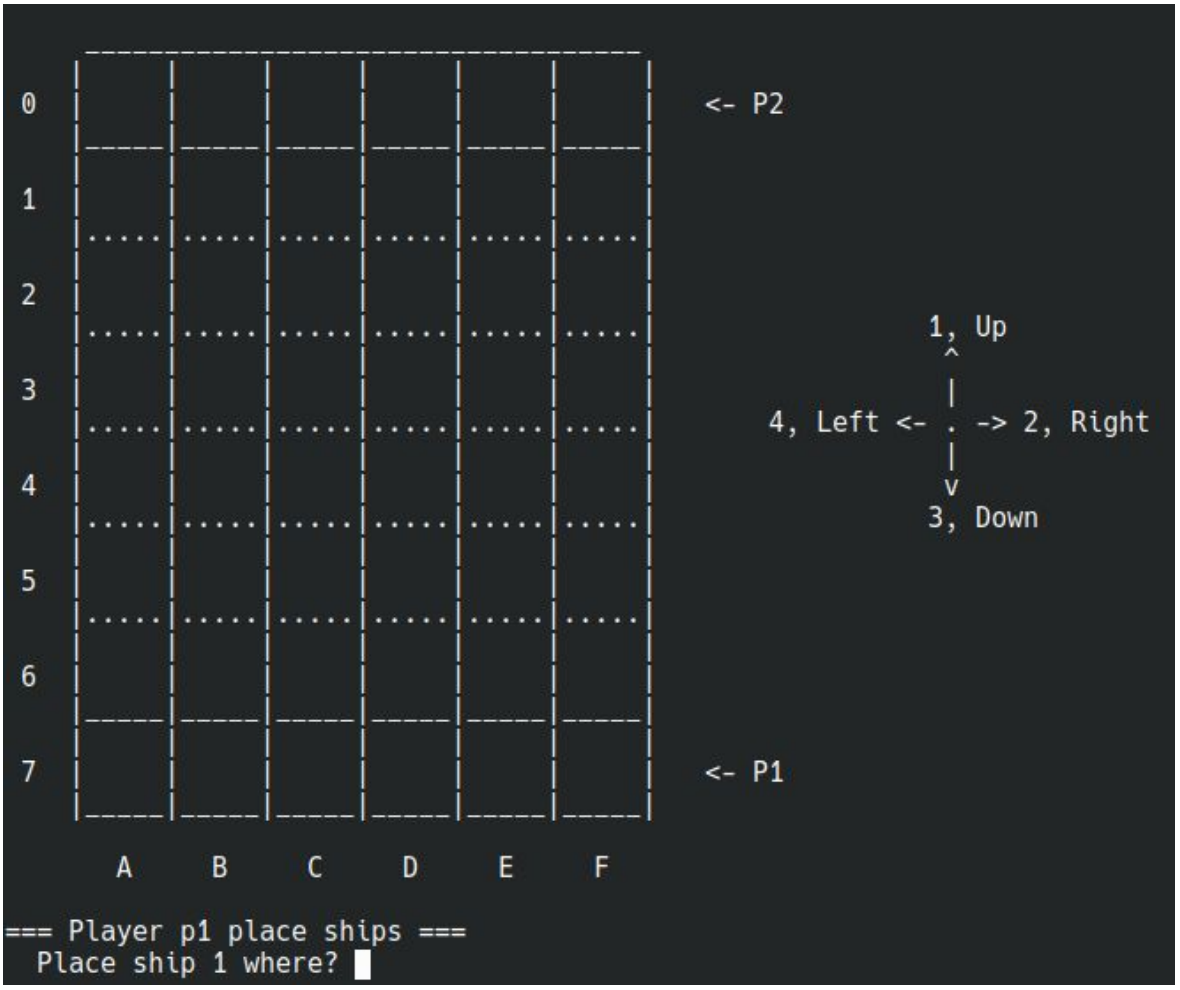

Figura 17: Jogador p1 a colocar as naves

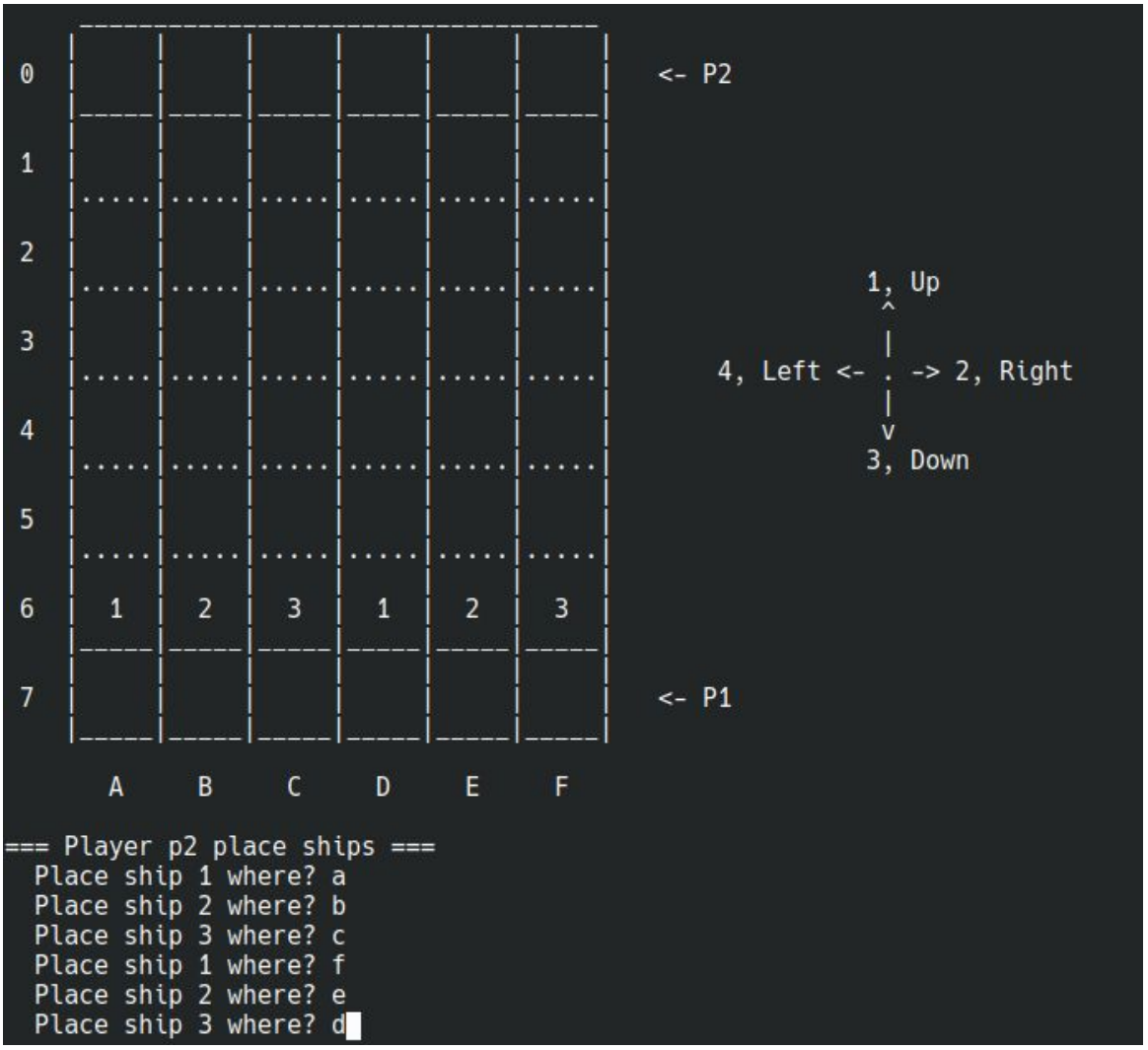

Figura 18: Jogador p2 a colocar a sua última nave após p1 já ter colocado as suas

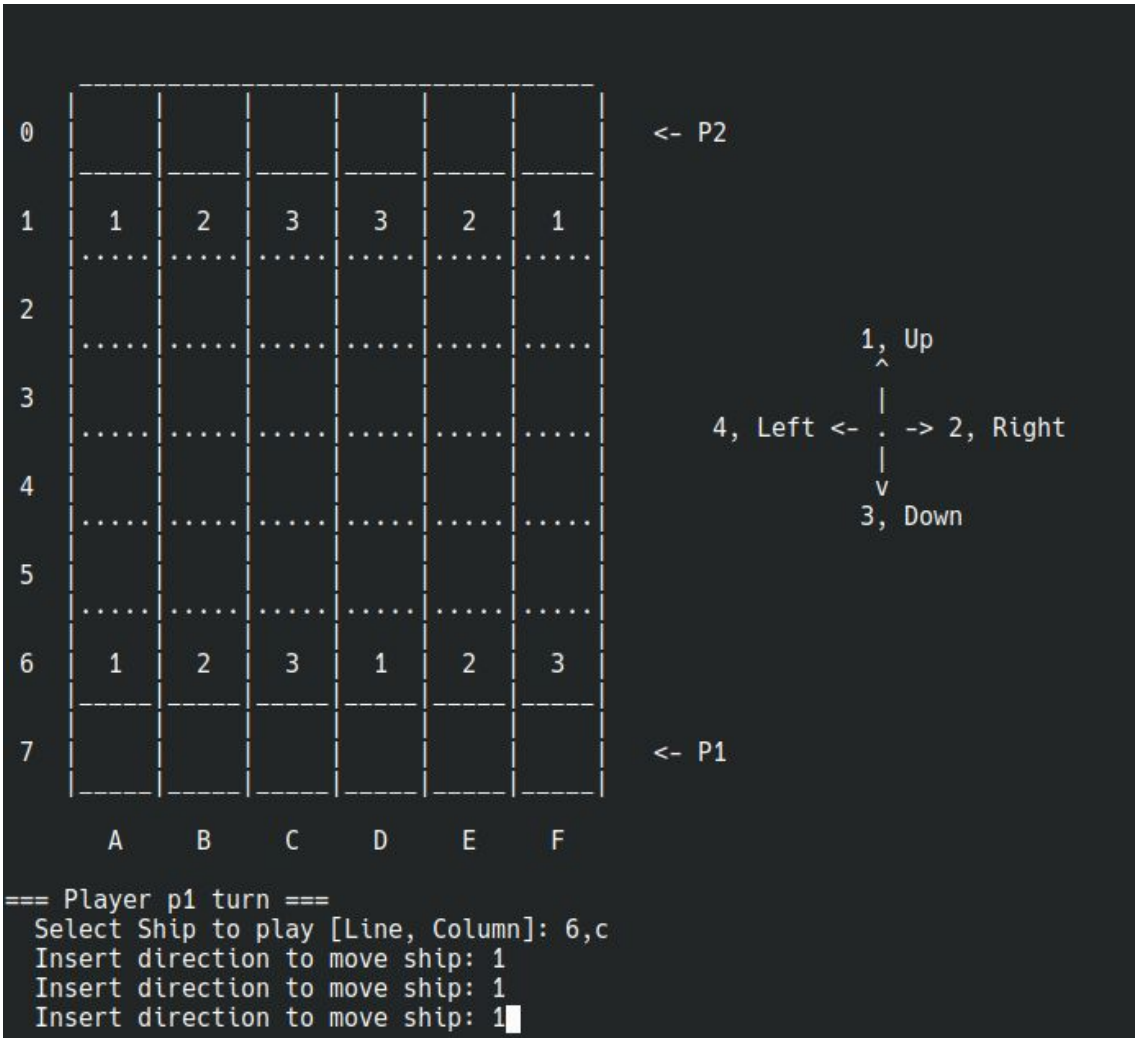

Figura 19: p1 a movimentar uma nave

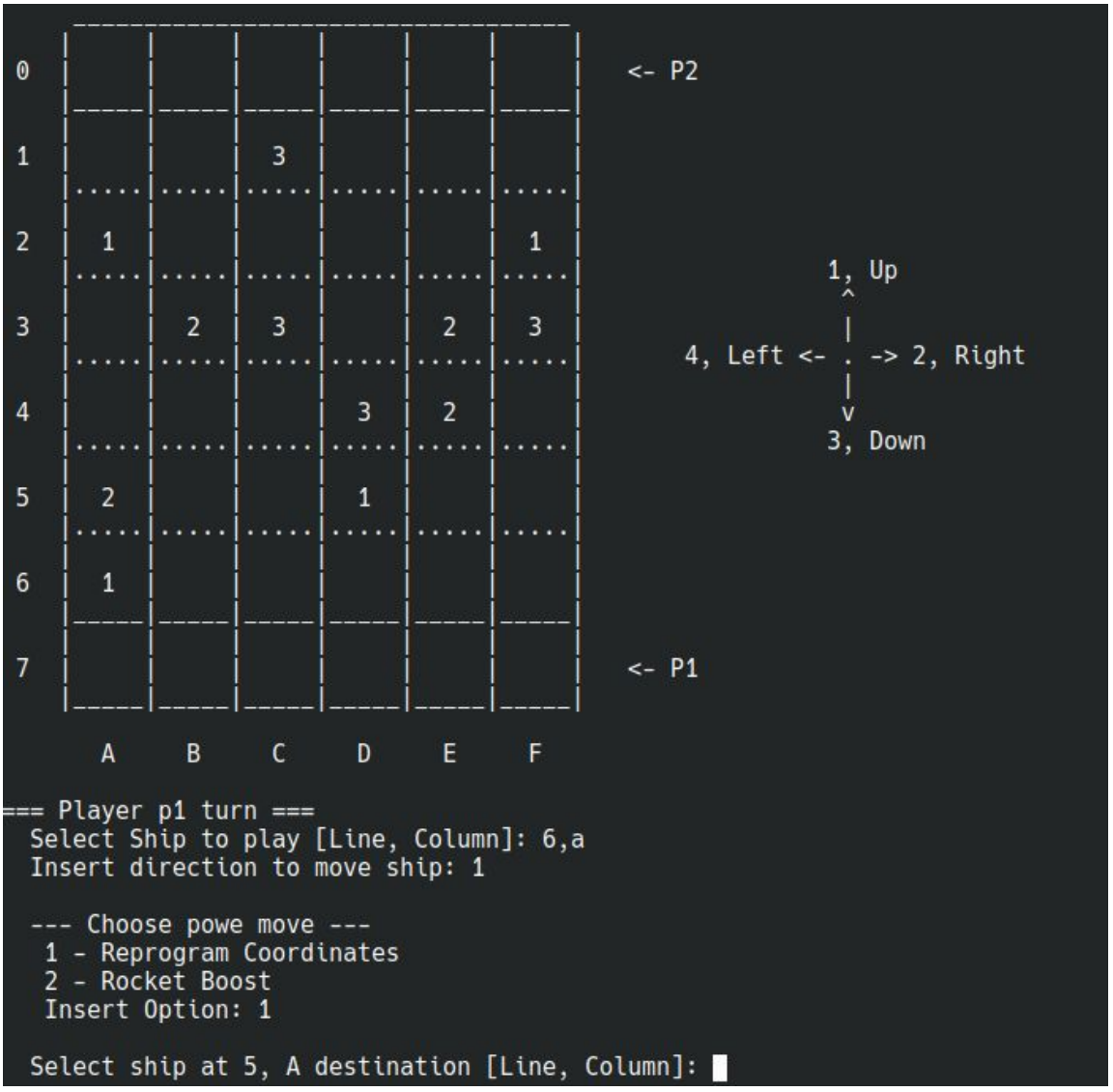

Figura 20: Menu de escolha das acções especiais e *Reprogram Coordinates*

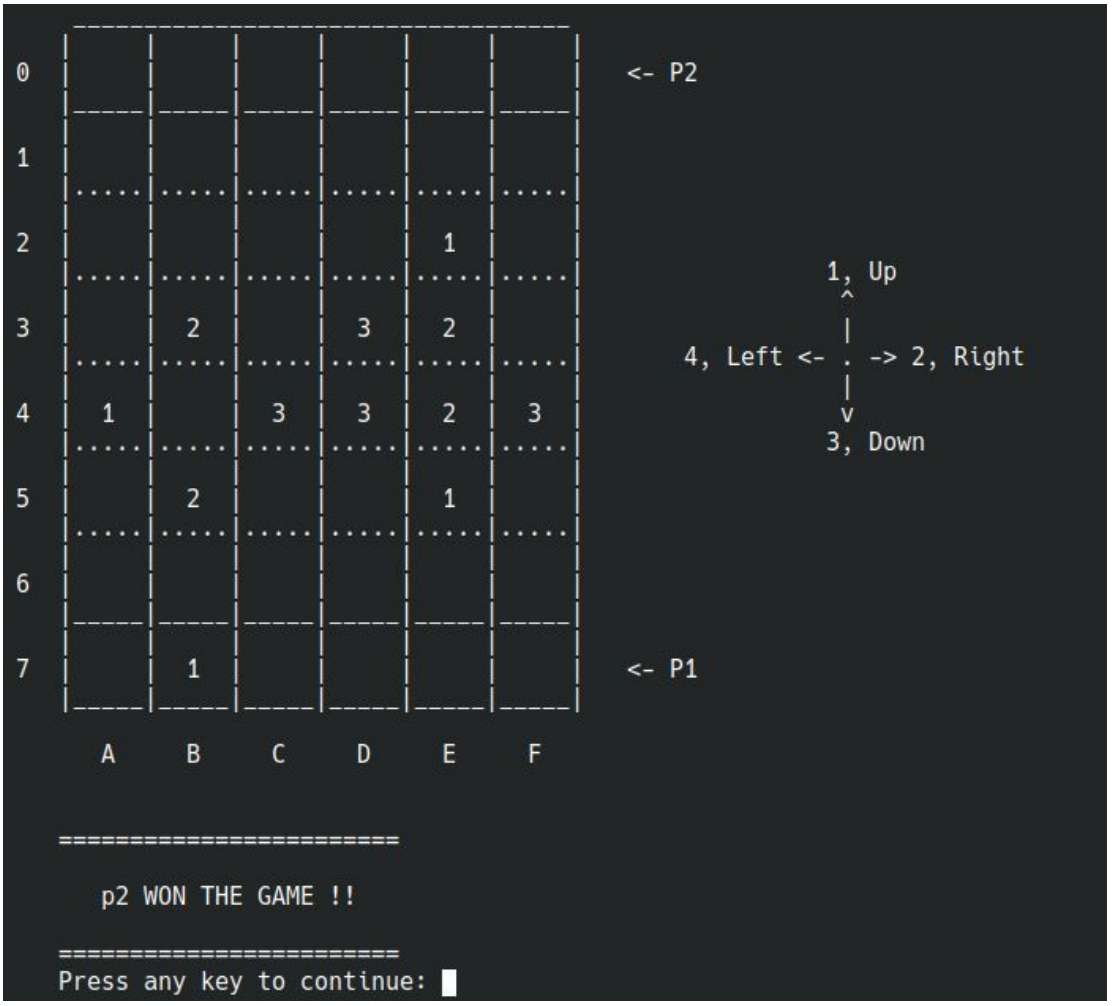

Figura 21: Vitória do jogador p2