U.PORTO
FEUP FACULDADE DE ENGENHARIA
UNIVERSIDADE DO PORTO

Mestrado Integrado em Engenharia Informática e Computação

Programação em Lógica

Relatório Final

Professores:

Rui Carlos Camacho de Sousa Ferreira da Silva
Henrique Daniel de Avelar Lopes Cardoso
Daniel Augusto Gama de Castro Silva

Grupo Small_Star_Empires_2:

Catarina Alexandra Teixeira Ramos, up201406219
Inês Isabel Correia Gomes, up201405778
Turma 5

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

13 de novembro 2016

1. Introdução

O seguinte projeto foi realizado no âmbito da cadeira de *Programação em Lógica* do 3º ano 1º semestre do *Mestrado Integrado em Engenharia Informática e Computação* da Faculdade de Engenharia da Faculdade do Porto.

Este trabalho tem como principal objetivo a elaboração um jogo de tabuleiro para dois jogadores caracterizado pelo tipo de tabuleiro e peças, pelas regras de movimentação das peças e pela condição de terminação do jogo. O jogo deve ser implementado na linguagem Prolog e deve permitir três modos de utilização: Humano vs Humano, Humano vs Computador e Computador vs Computador. O computador deve ter incluído pelo menos dois níveis de jogo e deve ser implementada uma interface adequada para o utilizador.

O relatório aqui apresentado está estruturado em cinco partes, sendo a primeira a introdução, a segunda, algumas informações sobre o jogo, como as sua regras, a terceira, a sua lógica, a quarta, a interface utilizada e por último, a quinta, as conclusões. A lógica do jogo está dividida em várias seções que variam desde a representação mais básica do jogo (a sua representação interna), até à inteligência artificial, passando pelas representações do tabuleiro e jogador, execução e validação de jogadas e finalização do jogo.

2. O jogo 'Small Star Empires'

Small Star Empires é um jogo estratégico de tabuleiro com peças hexagonais que tem como objetivo colonizar sistemas de modo a obter a pontuação mais alta. Cada peça do tabuleiro é um sistema onde o jogador se pode deslocar com as suas naves, e colonizar com as suas colónias/estação de trocas, conforme as regras.

De seguida temos uma breve descrição dos sistemas e peças do jogador, incluindo o número de peças iniciais e as pontuações obtidas por sistema ou peça.

Sistemas:

Vazio	Sistema vazio. O Jogador não recebe pontos por colonizar este sistema.
Estrela	Sistemas constituídos por 1 a 3 planetas. O jogador recebe o número de pontos equivalente ao número de planetas do sistema.
Nébula	Sistemas Vermelhos ou Azuis. O jogador recebe pontos pelo número de nébulas dominadas de uma só cor. Uma nébula é 1 ponto, 2 são 4 pontos e 3 são 7 pontos.
Blackhole	O jogador não pode dominar e/ou mover para este sistema, pelo que também não recebe qualquer pontuação.

Figura 1 - Vários tipos de Sistemas

Peças:

Nave	Meio de deslocamento do jogador que permite colonizar novos sistemas. Cada jogador terá em sua posse 4 naves.
Colónia	Peças usadas na colonização e proclamação dum sistema. Cada jogador possui 16 colónias.
Estação de trocas	Peças com a mesma funcionalidade das colónias. Por cada estação de trocas o jogador recebe pontos iguais ao número de inimigos em peças adjacentes. Cada jogador possui 4 estações de troca.

Os jogadores devem também seguir algumas regras:

- os jogadores movem-se por turnos, e em cada turno devem mover uma nave e colocar um controlo (colónia ou estação de trocas);
- um jogador só pode deslocar-se numa das 6 direções possíveis, em linha reta, o número de casas desejado;
- durante o movimento o jogador pode:
    - mover-se para qualquer sistema não ocupado;
    - passar por sistemas sob o seu controlo.
- durante o movimento o jogador não pode:
    - mover para um sistema sob o seu controlo;
    - passar por sistemas inimigos;
    - mover ou passar por blackholes.

O jogo termina quando ou nenhum dos jogadores tem jogadas possíveis.

3. Lógica do jogo

a. Representação do Estado do Jogo

O estado do jogo pode ser representado internamente e externamente. Internamente corresponde aos dados interpretados pelo computador. Externamente corresponde à visualização pelo utilizador.

Na representação interna do **tabuleiro**:

- 0 : corresponde aos espaços vazios (só serve para indentação);
- [x, y, z] : corresponde a três id's
    - x : existem 7 id's correspondentes a cada sistema da seção 1
    - y : id do player conjugado com a respectiva colónia/sistema de troca.
        - 0 e 1 : jogador 1;
        - 2 e 3 : jogador 2;
        - 0 e 2 : colónias;
        - 1 e 3 : sistema de trocas.
    - z : número de naves nessa célula.

Na representação externa do **tabuleiro**: [tipo de sistema, player, nº de naves]

- Tipo de sistema:
    - Sx : sistema de x planetas com x entre 0 e 3;

- Nx : nébula de cor x com x igual a r (vermelho) ou b (azul);
- B : blackhole;
- H : homeworld;

- Player : 1C/T ou 2C/T consoante é o jogador 1 ou 2 e se colocou uma colónia ou sistema de troca.

```
board(1,[[0,[1,-1,0],[5,-1,0],[2,-1,0],[0,-1,0]],
         [[4,-1,0],[6,0,4],[3,-1,0],[7,-1,0],[3,-1,0]],
         [0,[2,-1,0],[0,-1,0],[4,-1,0],[1,-1,0]],
         [[7,-1,0],[2,-1,0],[5,-1,0],[6,2,4],[0,-1,0]],
         [0,[3,-1,0],[5,-1,0],[0,-1,0],[4,-1,0]]]).
```

Figura 2 - representação a matriz interna do estado inicial do jogo.

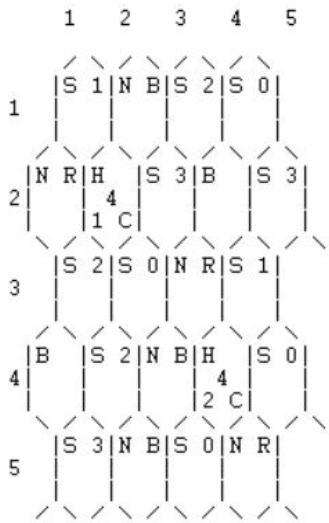

Figura 3 - representação externa do estado inicial do jogo.

```
board(5,[[0,[1,2,0],[5,3,0],[2,0,1],[0,2,1]],
         [[4,2,1],[6,0,0],[3,0,0],[7,-1,0],[3,2,1]],
         [0,[2,1,0],[0,0,0],[4,0,1],[1,0,1]],
         [[7,-1,0],[2,2,0],[5,2,0],[6,2,0],[4,3,1]],
         [0,[3,-1,0],[5,-1,0],[0,-1,0],[4,-1,0]]]).
```

Figura 4 - representação interna do estado de final do jogo.

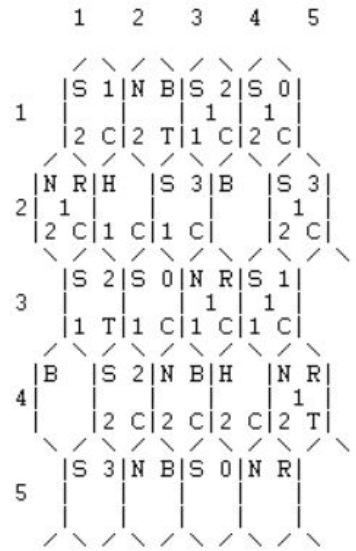

Figura 5 - representação externa de um possível estado final do jogo.

b. Representação do Jogador

Por uma questão de eficiência, o jogador é representado sob a forma de uma lista com todas as suas características e posses. Desta forma, é mais rápido e fácil efetuar a contagem dos pontos e obter a lista de movimentos possíveis conforme a posição das naves.

Representação interna do **jogador:** [equipa, tipo, estação de trocas, colónias, naves]

- Equipa:
  - 1 - Equipa Azul
  - 2 - Equipa Vermelha
- Tipo:
  - ‘C’ - Computador
  - ‘H’ - Humano
- Estação de Trocas:
  - Lista de posições das estações de trocas dominadas pelo jogador.
- Colónias:
  - Lista de posições das colónias dominadas pelo jogador.
- Naves:
  - Lista de posições das naves do jogador.

c. Visualização do tabuleiro

Nestre trabalho foram usados dois predicados para representar as células do tabuleiro:

- ***dominion****(ID, Equipa, Colonia_Trade).*
- ***systemType****(ID, Nome_sistema, Propriedade).*

O **dominion** tem um id associado bem como a equipa do jogador (1 ou 2) e se essa célula está ocupada por uma colónia ou estação de troca. O **systemType** é constituído por um ID, o nome da célula e a propriedade associada (por exemplo, na nébula tem a sua cor). Além destes dois predicados ainda é representado por um número ‘N’ que representa o número de naves dessa célula.

Para imprimir o tabuleiro são usados os seguintes predicados:

- **displayBoard**([L1 | L2]).
    - **displayColumnN**(L1, N).
    - **displayTopLine**(L1).
    - **displayMatrix2D**([L1 | L2] ,Row).
        - **displayLine**(L1 , Row).
            - **displayLine1**([L1 | L2]).
                - **displayInfo1**([IDs | [IDp | [N | [ ]]]])
                - **systemType**(IDs, _A, _B).
                - **displaySystem**(IDs).
            - **write**(Row).
            - **displayLine2**([L1 | L2]).
                - **displayInfo2**([IDs | [IDp | [N | [ ]]]]).
                - **write**(N).
            - **displayLine3**([L1 | L2]).
                - **displayInfo3**([IDs | [IDp | [N | [ ]]]]).
                - **dominion**(IDp, _A, _B).
                - **displayDominion**(IDp).
            - **displayBottomLine**([L1 | L2]).

O **displayBoard** chama recursivamente cada predicado abaixo com o número do tabuleiro escolhido. **boardInfo** apenas disponibiliza um texto de ajuda à interpretação do tabuleiro. **board** recebe o número do tabuleiro e procura-o na lista de factos do predicado **board**. De seguida, **displayTopLine** começa por desenhar a parte superior da célula, e o **displayMatrix2D** passa à interpretação do tabuleiro, fazendo display linha a linha através do predicado **displayLine**. O displayLine é dividido em 3 displays: **displayLine1** que representa a informação sobre o sistema; **displayLine2** que representa a informação sobre o player; **displayBottomLine** que desenha a parte de baixo da célula.

Figura 6 - Representação das propriedades das células do tabuleiro visualmente.

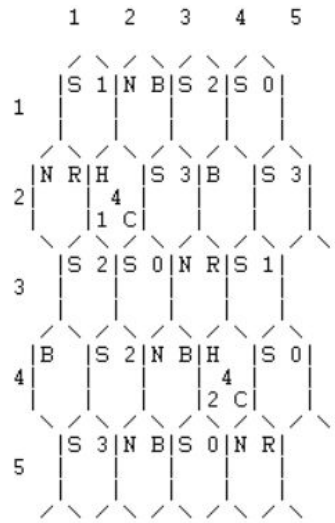

Figura 7 - Representação do tabuleiro pequeno do jogo inicial.

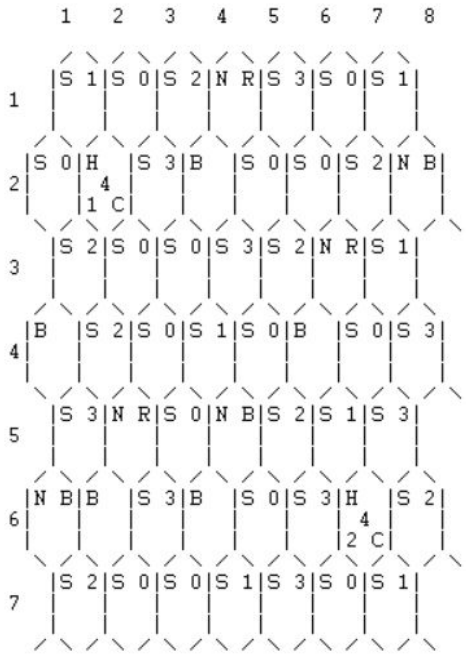

Figura 8 - Representação do tabuleiro médio do jogo inicial.

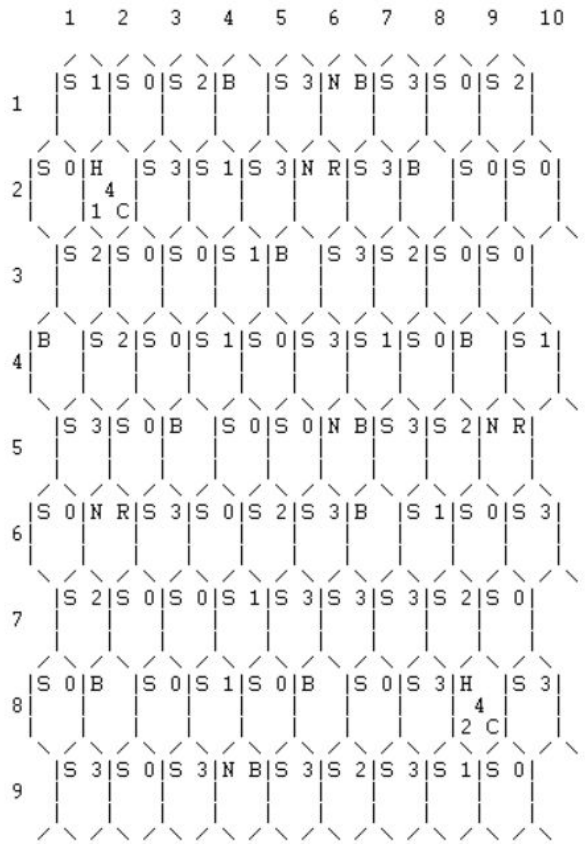

Figura 9 - Representação do tabuleiro grande do jogo inicial.

d. Lista de jogadas válidas

Para obter uma lista de jogadas válidas usamos ***getPossibleMoves(+Board ,+Playerl, -AllMoves)***, que retorna *AllMoves*, uma matriz de posições possíveis para onde o movimento é válido. A chamada de predicados relevantes a partir de *getPossibleMoves* é :

getAdjFreeCells(+Board, +Row,+Column, +Team, -Cells)
getAllFreeCellsInDirection(+Board, +Row, +Column, +Direction, +Team, +Tmp, -Cells)
freeCellInDirection(+Board,+Team,+Rowl,+Columnl,+Direction,-RowF,-ColumnF)
getCellInDirection(+Board,+Rowl,+Columnl,+Direction,-RowF,-ColumnF,-Cell)

O predicado ***getCellInDirection*** retorna a coluna, a linha e a célula da célula adjacente à posição de uma célula. Esta é chamada em ***freeCellInDirection*** que retorna a coluna e linha da próxima célula vazia sem passar por cima de células do jogador oposto ou de blackholes. Por sua vez, este é chamado recursivamente em ***getAllFreeCellsInDirection*** para obter a lista de todas as posições livres numa direção até encontrar o limite do tabuleiro, um blackhole ou uma célula dominada pelo jogador adversário. Com este predicado, ***getAdsFreeCells*** chama-o seis vezes, para as seis direções diferentes obtendo as listas de posições livres e junta-as. Por fim, este último predicado é chamado em ***getAdjFreeCells*** para cada nave na lista de naves do jogador.

Pode se concluir, portanto, que cada linha i da matriz *AllMoves* do *getPossibleMoves* corresponde aos movimentos possíveis para a nave na posição i da lista de naves do jogador.

e. Execução de jogadas

Uma jogada começa com ***play(+BoardI, +Nivel, +Team, +Player1I, +Player2I, -Player1F, -Player2F, -BoardF)*** que chama ***turn(+BoardI, +Nivel, +PlayerI, -BoardF, -PlayerF)*** onde efetua o turno do jogador da equipa Team e no fim chama ***play*** para o jogador da equipa adversária com o novo tabuleiro e o novo jogador. Desta forma, o predicado ***play*** é recursivo alternando as jogadas entre os ambos jogadores.

O predicado ***turn*** começa por atualizar as naves do jogador para só ficar com as que ainda se podem movimentar usando o predicado ***updateValidShips(+Board, +PlayerI, -PlayerF)***. De seguida obtém a matriz de movimentos possíveis com ***getPossibleMoves(Board,PlayerI,AllMoves)*** e verifica se o jogo chegou ao fim. Caso não seja o caso, verifica o predicado ***make_move(+BoardI, +Nivel, +AllMoves, +PlayerI, -BoardF, -PlayerF)*** que efetua a escolha de um movimento em ***movement(+Nivel, +Board, +Player, +AllMoves, -RowI, -ColumnI, -RowF, -ColumnF)*** e valida-o. Depois dessa validação, coloca uma peça de controlo (colónia ou estação de trocas) com ***addControl(+Nivel, +BoardI, +PlayerI, +Row, +Column, -BoardF, -PlayerF).*** Por fim, efetua o deslocamento da nave em (*RowI,ColumnI*) para (*RowF,ColumnF*), com ***setShip(+Board, +Row, +Column, +ShipsAdd, -Final)*** e a atualização da nave na lista de naves do jogador com ***playerSetShip(+PlayerI, +PosI,+ PosF, -PlayerF)***.

No predicado ***movement*** o tipo de escolha das coordenadas diverge conforme o tipo de jogador (Computador ou Humano) e a validação é feita usando o ***validMove(+AllMoves, +PlayerI, +RowI, +ColumnI, +RowF, +ColumnF)***. Este predicado verifica se existe uma nave com a posição (*RowI,ColumnI*) e em que índice i da lista de naves do jogador se encontra. Depois, verifica se existe a posição (*RowF,ColumnF*) na lista da linha i da matriz *AllMoves*. Se a posição existir, então o movimento é válido, visto que, o índice da linha da matriz *AllMoves,* corresponde às posições válidas para a nave desse índice na lista de naves do jogador.

No predicado ***addControl*** obtém-se o tipo de peça a colocar, conforme o tipo de jogador, e coloca o domínio com o predicado ***setDominion(+Board, +Team, +Row, +Column, +Type, -Final)*** guardando a posição do domínio na lista correspondente no jogador.

f. Avaliação do tabuleiro

Aquando a análise do tabuleiro, para efetuar a melhor jogada, inconscientemente cada pessoa faz a sua análise do tabuleiro. Essa análise pode ser passada para a linguagem Prolog, promovendo a inteligência artificial do nosso jogo. O objetivo é priorizar as jogadas que dão maior pontuação ao jogador. Para tal, é usado o predicado

getPossibleMovesValued(+Board,+Player,+AllMoves,-ValuedMoves)

que recebe o tabuleiro, o jogador e a matriz de todas as jogadas possíveis, e devolve a matriz correspondente dos "valores" dessas posições. Por sua vez este predicado usa

setValuedMoves(+Board,+Player,?AllMoves,?TempList,-ValuedMoves)

que preenche a matriz *ValuedMoves* de acordo com a seguinte metodologia:

- percorre a matriz *AllMoves* (matriz de posições de jogadas possíveis):
    - para cada posição dessa matriz, retira a posição dessa célula no tabuleiro;
    - avalia a célula no tabuleiro e atribui-lhe um valor:
        - 0 : sistema vazio;
        - 1 : Sistema do tipo *Star* com um planeta;
        - 2 : Sistema do tipo *Star* com dois planetas ou nébula (caso o jogador só tenha uma dessa cor, nesse momento);
        - 3 : Sistema do tipo *Star* com três planetas;
        - 4 : Nébula cuja cor o jogador já colonizou.

A atribuição destes valores baseia-se nos pontos obtidos diretamente pelas jogadas efetuadas. A estratégia passa por escolher as células que dão, neste momento ou no futuro, mais pontos ao jogador. As células com valor 4 são de alta prioridade, enquanto as células de valor 0 são de baixa prioridade.

g. Final do jogo

O jogo acaba quando não houver mais jogadas possíveis pela parte do jogador que efetua o turno. Sendo este o caso, esta verificação é feita dentro do predicado ***turn(+BoardI, +Nivel, +PlayerI, -BoardF, -PlayerF)*** após a verificação do predicado ***getPossibleMoves(+Board ,+PlayerI, -AllMoves).*** Como fazemos a atualização das naves do jogador retirando da lista as que não conseguem efetuar mais movimentos, a matriz *AllMoves* ficará vazia se não existirem mais nenhuma nave disponível. Desta forma, a verificação é feita com *AllMoves* \= *[ ],* que se for falsa então o jogador sai de ***turn*** e passa para o ***play*** seguinte. Este apenas iguala o tabuleiro e jogadores iniciais aos finais, procedendo na verificação dos predicados no predicado inicial ***game***.

Depois de acabar o jogo, é necessário verificar quem o ganhou. Para tal é usado o método ***winner(+Board,+P1,+P2)*** que recebe o tabuleiro e os dois jogadores e procede à contagem dos pontos. No final, quem tiver mais pontos vence o jogo. Se tiverem os mesmos pontos, é um empate.

A atribuição dos pontos faz-se em duas fases: contagem dos pontos pessoais e atribuição de 3 pontos extra ao jogador com território (colónias e estações de troca) maior. O número de pontos atribuídos consoante sistema está definido na secção 2 deste relatório.

A contagem dos pontos pessoais segue a seguinte metodologia:

- é construída uma lista com as posições do território do jogador;
- a lista é percorrida, e para cada posição é verificada a sua célula, se:
    - célula do tipo *Star,* acrescenta os pontos correspondentes;
    - célula do tipo *Nebulae,* incrementa o contador correspondente;
    - célula preenchida com uma estação de troca, verifica todas as células adjacentes e acrescenta um ponto por cada inimigo encontrado;
- no final, acrescenta os pontos de acordo com os acumuladores das células tipo *Nebulae*.

A regra que faz esta verificação é

countPoints(+Board,?Territory,?AccPoints,-Points,?AccB,?AccR)

que recebe o Board, uma lista com as posições do território, os pontos acumulados, os acumuladores AccB e AccR que são atualizados a cada iteração, e devolve os pontos contados.

Para auxiliar esta regra, também são usados os predicados

getTradePoints(+Board,+R,+C,+DominionID,+Points,-FinalPoints)

getSystemTypePoints(+SystemType,,-Points,? AccB,?AccR, -NewAccB,-NewAccR)

getNebulaePoints(+Points, +Acc, -FinalPoints)

correspondentes aos “pontos a branco” desta secção.

Para a atribuição dos 3 pontos extras é usado o predicado

biggestTerritoryPoints(+Length1,+Points1,-NewPoints1,+Length2,+Points2, -NewPoints2)

que recebe o tamanho e os pontos atuais de cada jogador, e devolve os novos pontos dos 2 jogadores.

Calculados os pontos, pode ser então escolhido o jogador pelo predicado

chooseWinner(+Board,+P1,+Points1,+P2,+Points2)

que recebe o tabuleiro, os pontos e os jogadores, e consoante a maior pontuação, imprime o resultado adequado.

h. Jogada do Computador

A jogada do computador é constituída por dois níveis de duas dificuldades, fácil e difícil.

Para processar estes níveis, o computador utiliza o mesmo ciclo de jogo do humano (descrito na secção *e)* do presente relatório), apenas diferindo nas seguintes regras:

movement(+Level,+Board,+Player,+AllMoves,-RowI,-ColumnI,-RowF,-ColumnF)

addControlAux(+Level,+Player,+Board,+Row,+Column,-Type)

O predicado *movement* recebe o nível, o tabuleiro, o jogador, e a matriz com todas as jogadas possíveis de efetuar, e retorna a posição (linha e coluna) inicial e final do computador. Já o *addControlAux* recebe o nível, jogador, tabuleiro linha e coluna e retorna o tipo de controlo a colocar (colónia ou estação de troca).

Para o nível 1, o computador utiliza a seguinte metodologia:

- escolhe uma nave aleatória (dentro das naves disponíveis) para mover;
- dentro das jogadas possíveis dessa nave, escolhe uma aleatoriamente.

Com este método, a posição inicial do jogador será a posição da nave escolhida e a final a posição aleatória da matriz *AllMoves.* As regras associadas a estes pontos são:

getRandShip(+Player,+AllMoves,-RowI,-ColumnI,-ShipMoves)

getRandMove(+ShipMoves,-RowF,-ColumnF)

Depois de movimentar a sua nave, o jogador deve estabelecer controlo sobre essa célula escolhendo uma colónia ou estação de troca. Para tal o computador usa o predicado:

chooseType(+Level,+Board,+Player,+AdjCells,+MyTeam,-Type)

chamado pelo predicado ***addControlAux.*** O seu objetivo é escolher uma colónia ou estação de troca dependendo do número de inimigos adjacentes. Se não houver inimigos adjacentes, ou já tiver colocado as 4 estações de trocas, o computador coloca uma colónia. Se o nível escolhido for 1, e tiver pelo menos um inimigo adjacente, coloca uma estação de trocas. Se o nível escolhido for o 2, só coloca estação de trocas caso existam pelo menos 2 inimigos em células adjacentes.

Para o nível 2, o computador utiliza a seguinte metodologia:

- calcula uma matriz (*ValuedMoves*) equivalente à matriz AllMoves com os "valores" das posições dessa matriz;
- procura a posição (linha e coluna da matriz ValuedMoves) com o "valor" mais alto;
- tendo em conta a posição devolvida da matriz, calcula qual o barco a movimentar (o que dá a posição inicial), e usando essa posição na matriz AllMoves encontra o valor da posição final.

Com este método, o jogador escolhe sempre a melhor jogada entre todas as jogadas possíveis. Para escolher a melhor jogada, é usada a regra

getBestMove(+ValuedMoves,-Ship,-Pos)

que recebe a matriz *ValuedMoves* (descrita anteriormente na secção f) do presente relatório), e devolve a linha e coluna da mesma matriz, que corresponde ao "valor" mais alto.

4. Interface com o Utilizador

O jogo inicializa com **game.** podendo no menu inicial escolher entre:

- Jogar (**1.**);
- Sair (**2.**).

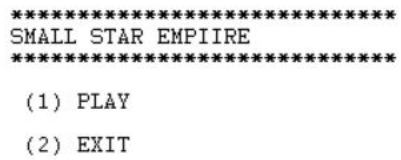

Figura 10 - representação do menu inicial do jogo.

Caso a opção escolhida seja jogar, irá avançar para o menu de opções do jogo em que teremos que escolher entre:

- O tamanho do tabuleiro :
  - Pequeno (**1.**);
  - Médio (**2.**);
  - Grande (**3.**).
- O modo de jogo:
  - Humano vs Humano (**1.**);
  - Humano vs Computador (**2.**);
  - Computador vs Computador (**3.**).
- O nível (caso o modo seja 2 ou 3):
  - Fácil (**1.**);
  - Difícil (**2.**).

Figura 11 - representação de um exemplo do menu para escolha das opções do jogo.

Se algum destes valores não estiver dentro do intervalo de valores aceites, serão pedidos valores novos.

Após a escolha das opções do jogo, o jogo irá iniciar. A cada turno será feito o display:

- Da equipa que está a efetuar o turno;
- De informações sobre a constituição das células do tabuleiro;
- Do tabuleiro;
- De uma lista TRADES com as posições das estações de trocas do jogador;
- De uma lista COLONIES com as posições das colónias do jogador;
- De uma lista SHIPS com as posições onde se encontram as naves válidas do jogador;
- De uma lista POSSIBLE MOVES com as posições de jogadas possíveis para cada nave pertencente ao jogador;

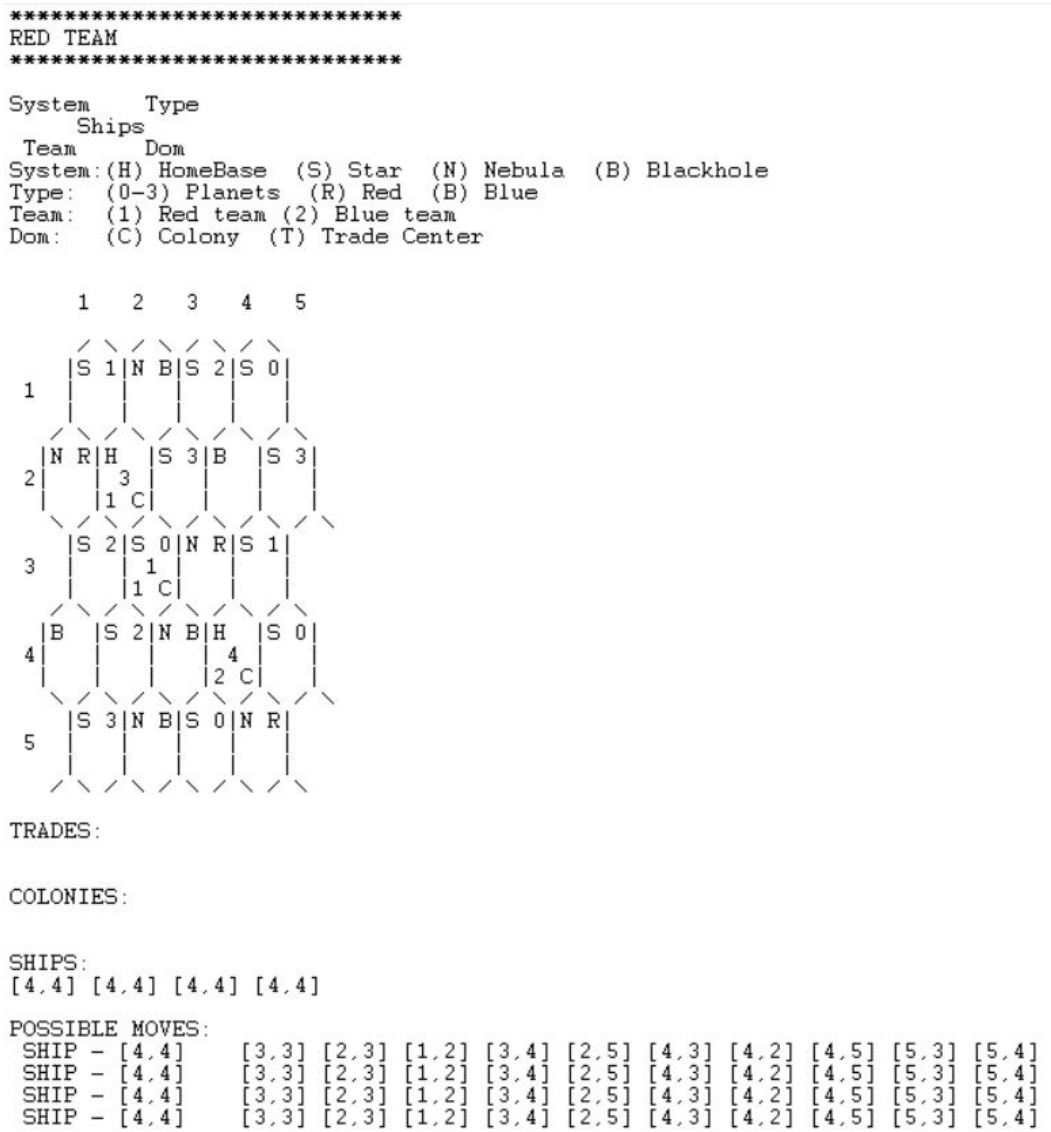

Figura 12 - representação de um exemplo de um turno.

Caso o turno pertença a um jogador humano, irá perguntar quais são as coordenadas da célula de partida e as de final e qual é o tipo de domínio que quer colocar:

- Estação de Trocas (**'T'.**);
- Colónia (**'C'.**).

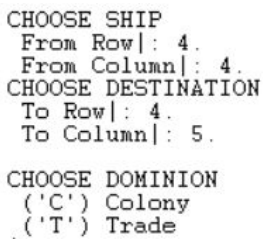

Figura 13 - representação de um exemplo da escolha de coordenadas e do domínio por um jogador do tipo humano.

Se a escolha das coordenadas não for válida ou se a escolha do domínio não for aceite, irá voltar a perguntar estas informações até ter valores válidos.

No final do jogo será feito o display:

- De uma header a sinalizar o final do jogo com “Game Over”;
- Do tabuleiro final;
- Da atribuição da vitória ao jogador vencedor;
- Dos pontos de ambos os jogadores.

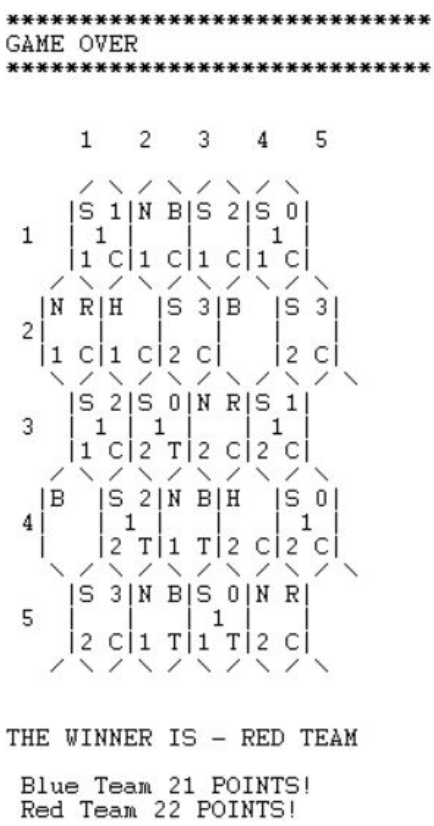

Figura 14 - representação de um exemplo de um final de jogo.

Após o final do jogo há ainda a possibilidade de voltar ao menu inicial e jogar novamente.

5. Conclusões

Com este trabalho podemos concluir que as linguagens de programação lógica são muito úteis para o desenvolvimento de inteligência artificial e problemas no campo linguístico. Neste trabalho conseguimos criar com sucesso um pequeno *CPU Player* que simulava um jogador e aplicava pequenos conhecimentos de inteligência artificial.

Embora tenha sido um trabalho complexo, conseguimos cumprir todos os objetivos e concluir que graças à linguagem lógica a solução conseguiu ser muito menos complexa que uma linguagem, por exemplo, orientada a objetos.

Para melhor este trabalho poderíamos ter implementado um modo em que o utilizador poderia construir o seu próprio tabuleiro (possibilidade do jogo original). Também poderíamos ter implementado mais níveis de dificuldade com inteligência artificial mais complexa.