U.PORTO

FEUP FACULDADE DE ENGENHARIA
UNIVERSIDADE DO PORTO

Blockade

Relatório Final

Mestrado Integrado em Engenharia Informática e Computação

2016/2017

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

PROGRAMAÇÃO EM LÓGICA

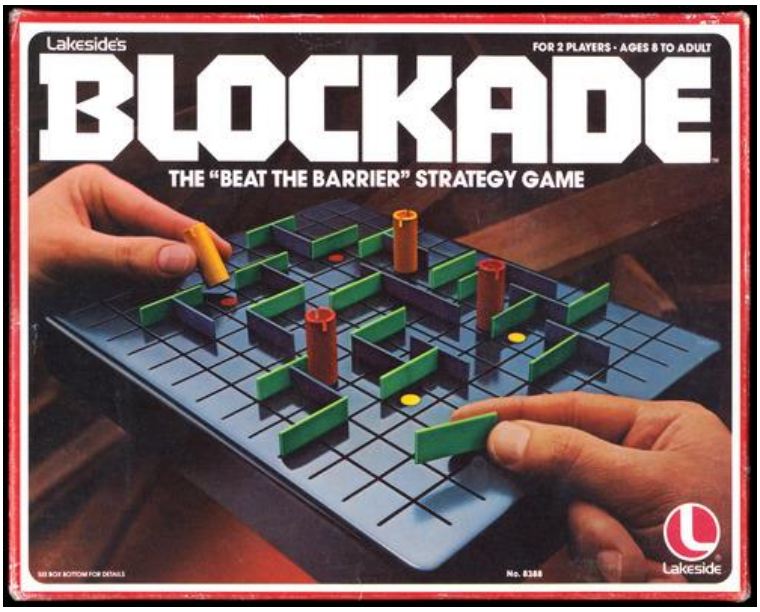
Grupo : 03
José Francisco Cagigal da Silva Gomes – up201305016
Margarida Xavier Viterbo – up201403205

12 de novembro de 2016

Blockade

Relatório Final

Resumo



O projeto aqui descrito consiste na implementação em *Prolog* do jogo de tabuleiro *Blockade*. O objetivo deste trabalho é cimentar os conhecimentos adquiridos nas aulas práticas e teóricas de Programação em Lógica, transformando um jogo de tabuleiro clássico num jogo de computador, utilizando uma interface muitos simples em texto, que mais tarde será implementada em 3D noutra unidade curricular.

Para esta transformação do jogo em formato digital foi utilizada uma abordagem simples: identificação das regras do jogo (lista de jogadas válidas, critério para declarar vencedor e terminar jogo); definição do tabuleiro e peças bem como a representação dos diferentes estados de jogo; verificação da existência de um caminho válido para a vitória durante todo o jogo.

Tudo isto foi implementado com sucesso, tendo sido mais difícil tratar da verificação de caminhos válidos, movimentação/colocação dos peões e paredes, a construção em modo texto de um tabuleiro manipulável (por parte do programa) e legível (por parte do utilizador) e jogo com CPU e níveis de dificuldade.

Índice

Introdução

Serve o presente relatório como apoio em suporte escrito ao projeto de implementação do jogo *Blockade* em *Prolog*, oferecendo uma perspetiva aprofundada do processo de criação do jogo nesta linguagem e funcionamento do mesmo.

Este trabalho apresenta-se como uma excelente forma de consolidar e aprofundar conhecimentos na área da Programação em Lógica, bem como desenvolver as capacidades de pensamento lógico de cada aluno.

De todos os temas propostos foi escolhido o *Blockade*, por se apresentar como um problema de decisão e previsão, sendo simultaneamente intuitivo e com regras fáceis de perceber. Muito sucintamente, o objetivo do jogo será chegar primeiro com os pinos às casas de começo dos pinos do adversário, possuído 8 paredes que ajudam a dificultar esta tarefa ao adversário.

Este relatório encontra-se dividido nas seguintes secções:

- O jogo *Blockade* – explicação sucinta do jogo;
- Lógica do Jogo – representação do estado do jogo, visualização do tabuleiro, lista de jogadas válidas, execução de jogadas, avaliação do tabuleiro, final do jogo, jogada do computador;
- Interface com o Utilizador – descrição da forma de visualização dos diferentes estados do jogo;
- Conclusões;
- Bibliografia;
- Anexos;

Este artigo pretende desta forma documentar a abordagem que o grupo utilizou para atingir o resultado pretendido, explicando o problema em detalhe e a sua resolução, demonstrando os frutos obtidos e a visualização dos mesmos.

1. O Jogo Blockade

Blockade é o jogo de tabuleiro de estratégia para dois jogadores inventado por Mirko Marchesi e publicado por *Lakeside Industries* em 1975. É dirigido a pessoas com mais de 8 anos e requer apenas a faculdade de dedução. Demora menos de um minuto a preparar e o tempo de cada jogo é cerca de 20 minutos[1].

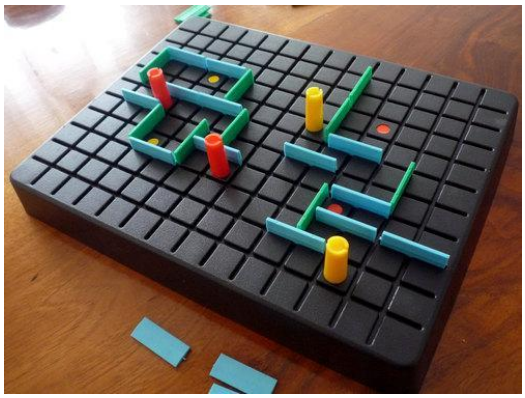


Figura 1- Blockade

Componentes:

Cada jogador deverá ter 2 peões, 9 paredes verdes (colocados sempre na vertical) e 9 paredes azuis (colocados sempre na horizontal).

Descrição:

No início os jogadores deverão colocar os peões nos sítios referenciados com a cor do peão (células [4,4] e [8,4] para um jogador e células [4,11] e [8,11] para o outro jogador) do tabuleiro (11x14).

O objetivo do jogo para cada jogador é colocar ambos os pinos nas posições iniciais dos pinos do adversário. Em cada jogada o jogador deverá mover um dos peões, no máximo, 2 posições (na vertical, horizontal ou uma combinação dos dois) e colocar uma parede (horizontal ou vertical) entre os espaços do tabuleiro e em qualquer sítio no mesmo, sendo que cada parede tem o comprimento de 2 espaços.

[1] https://en.wikipedia.org/wiki/Blockade_(board_game)

O objetivo da parede será bloquear os movimentos do adversário. Uma vez que os jogadores fiquem sem paredes continuam a jogar movendo apenas os peões. O primeiro a atingir o objetivo ganha[2].

Especificação das Regras:

- Os peões não podem passar para uma posição do tabuleiro, se entre a posição pretendida e a posição em que se encontra, estiver colocada uma parede.
- Cada jogador só pode mexer um peão por jogada duas posições (vertical, horizontal ou uma combinação das duas).
- Cada jogador só pode colocar uma parede por jogada.
- Um peão pode saltar por cima de outro peão que lhe esteja a bloquear o caminho.
- Não podem haver 2 ou mais peões na mesma posição.
- As paredes não podem ser atravessadas por outras paredes.
- Tem sempre de haver um caminho entre cada peão e cada posição inicial adversária.
- As paredes não podem ser recolocadas.

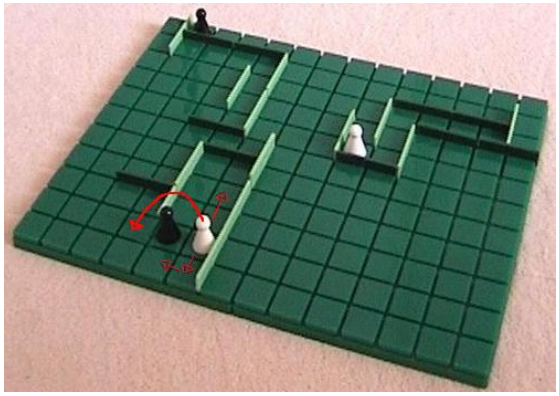


Figura 2 - Joagadas possíveis

[2] http://boardgamegeek.com/boardgame/2559/blockade

2. Lógica do Jogo

O código desenvolvido para executar este jogo encontra-se dividido por 4 ficheiros:

- ***Main****: inicio do programa, contém predicados principais que utilizam os restantes predicados necessários ao funcionamento do jogo;*

- ***Rules****: contém os predicados de verificação da movimentação e colocação de paredes, implementação das regras do jogo em prolog;*

- ***InOut****: predicados responsáveis pelo input e output de dados;*

- ***AlgCpu****: predicados responsáveis pelo jogo da máquina;*

2.1. Representação do Estado do Jogo

A representação do tabuleiro é feita através de uma lista de listas em que são usados os símbolos ‘a’ e ‘b‘ para desenhar as limitações células do tabuleiro, ‘c’ para representar as casas do tabuleiro e ‘p’ e ‘j’ para representar as posições iniciais dos peões dos jogadores.

```
board([[c,a,c,a,c,a,c,a,c,a,c,a,c,a,c,a,c,a,c,a,c],
        [b,b,b,b,b,b,b,b,b,b,b],
        [c,a,c,a,c,a,c,a,c,a,c,a,c,a,c,a,c,a,c,a,c],
        [b,b,b,b,b,b,b,b,b,b,b],
        [c,a,c,a,c,a,c,a,c,a,c,a,c,a,c,a,c,a,c,a,c],
        [b,b,b,b,b,b,b,b,b,b,b],
        [c,a,c,a,c,a,p,a,c,a,c,a,c,a,p,a,c,a,c,a,c],
        [b,b,b,b,b,b,b,b,b,b,b],
        [c,a,c,a,c,a,c,a,c,a,c,a,c,a,c,a,c,a,c,a,c],
        [b,b,b,b,b,b,b,b,b,b,b],
        [c,a,c,a,c,a,c,a,c,a,c,a,c,a,c,a,c,a,c,a,c],
        [b,b,b,b,b,b,b,b,b,b,b],
        [c,a,c,a,c,a,c,a,c,a,c,a,c,a,c,a,c,a,c,a,c],
        [b,b,b,b,b,b,b,b,b,b,b],
        [c,a,c,a,c,a,c,a,c,a,c,a,c,a,c,a,c,a,c,a,c],
        [b,b,b,b,b,b,b,b,b,b,b],
        [c,a,c,a,c,a,c,a,c,a,c,a,c,a,c,a,c,a,c,a,c],
        [b,b,b,b,b,b,b,b,b,b,b],
        [c,a,c,a,c,a,c,a,c,a,c,a,c,a,c,a,c,a,c,a,c],
        [b,b,b,b,b,b,b,b,b,b,b],
        [c,a,c,a,c,a,j,a,c,a,c,a,c,a,j,a,c,a,c,a,c],
        [b,b,b,b,b,b,b,b,b,b,b],
        [c,a,c,a,c,a,c,a,c,a,c,a,c,a,c,a,c,a,c,a,c],
        [b,b,b,b,b,b,b,b,b,b,b],
        [c,a,c,a,c,a,c,a,c,a,c,a,c,a,c,a,c,a,c,a,c],
        [b,b,b,b,b,b,b,b,b,b,b],
        [c,a,c,a,c,a,c,a,c,a,c,a,c,a,c,a,c,a,c,a,c]]).
```

Figura 3 - Definição do tabuleiro de jogo em Prolog

Esta definição produz o seguinte resultado:

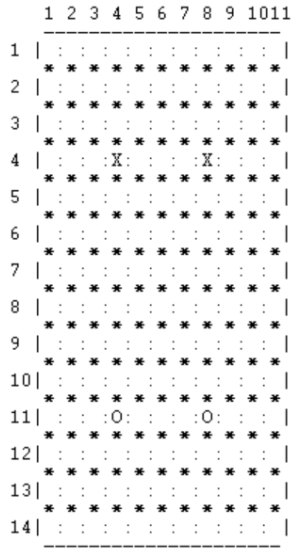


Figura 4 - Estado inicial do jogo

Após algumas jogadas poderíamos obter o seguinte tabuleiro:

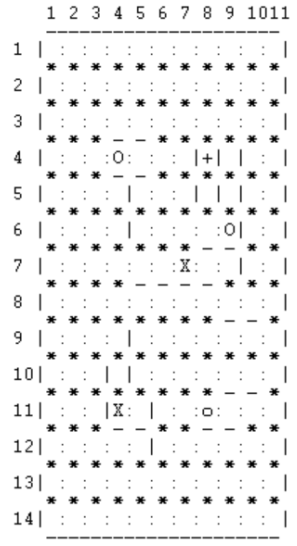


Figura 5 - Estado intermédio do jogo

Um panorama possível para o estado final do jogo seria o seguinte:

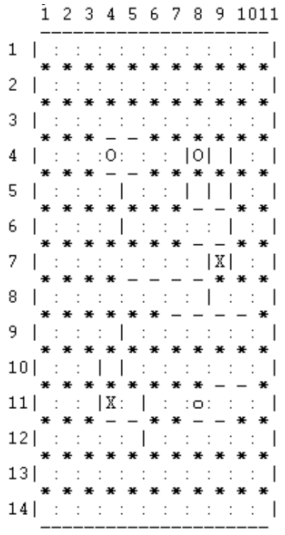


Figura 6 - Estado final do jogo

2.2. Visualização do Tabuleiro

Os predicados usados para a visualização do tabuleiro foram os seguintes:

- ***startDisplay(Board,X,Y)*:** *Board* é o tabuleiro de jogo, *X* e *Y* são a posição que vai ser imprimida no momento; esta função começa o desenho dos elementos acessórios ao tabuleiro e chama as funções que fazem o desenho do tabuleiro em si;
- ***display([L1|L2],X,Y):*** *[L1|L2]* é a lista de listas que representa o tabuleiro, *X* e *Y* são a posição que vai ser imprimida no momento;
- ***displayy([L1|L2],X,Y):*** é chamada pela função *display* para desenhar o conteúdo de uma linha do tabuleiro.
- ***translate(Char,X,Y,Result):*** *Char* é o caractere que está na matriz do tabuleiro na posição que está a ser escrita, *X* e *Y* dão as coordenadas desse posição e *Result* é o que vai ser impresso no ecrã.

```
startDisplay(Board,X,Y):-write('    1 2 3 4 5 6 7 8 9 1011'),nl,
    write('    ---------------------'),nl,display(Board,X,Y),write('    ---------------------'),nl.
display([],_,_).
display([L1|L2],X,Y):-(X mod 2=\=0,X2 is round((X+1)/2),write(X2),(X2>9,write('-'),write('|');write(' -'),
    write('|'));write('    ') ),displayy(L1,X,Y),(X mod 2=\=0,write('|');true),nl,X1 is X+1,display(L2,X1,Y).
displayy([],_,_).
displayy([L1|L2],X,Y):-translate(L1,X,Y,Value),write(Value),Y1 is Y+1,displayy(L2,X,Y1).

translate(Char,X,Y,Result):-pawn1([L3,L7]),pawn2([L4,L8]),pawn3([L5,L9]),pawn4([L6,L10]),
    (Char==w,Result ='- ';Char ==q, Result ='|';Char ==a, Result =':';Char ==b, Y\=12, Result ='* ';
    Char ==c, Result =' ';Char ==r, Result ='X';Char ==e, Result ='O';Char ==p,
    ((2*L3-1=:=X,2*L7-1=:=Y;2*L4-1=:=X,2*L8-1=:=Y),Result ='X';(2*L5-1=:=X,2*L9-1=:=Y;2*L6-1=:=X,2*L10-1=:=Y),
    Result ='O';Result ='+');Char ==j, ((2*L3-1=:=X,2*L7-1=:=Y;2*L4-1=:=X,2*L8-1=:=Y),
    Result ='X';(2*L5-1=:=X,2*L9-1=:=Y;2*L6-1=:=X,2*L10-1=:=Y),Result ='O';Result ='o')).
```

Figura 7 - Predicados para desenhar o tabuleiro

2.3. Lista de Jogadas Válidas

Os predicados que verificam se as jogadas são válidas são todos aqueles que verificam se a movimentação do peão é válida e se a colocação da parede é possível.

Para se colocar uma parede com sucesso, tem de se verificar se os seguintes predicados são cumpridos:

- ***availableWallHorizontal(Board,X,Y):*** recebe o tabuleiro e percorre todas as posições a partir de (X,Y), colocando numa variável dinâmica todas as posições do tabuleiro onde podem estar paredes horizontais;
- ***availableWallVertical(Board,X,Y):*** igual à função anterior mas para paredes verticais.

- ***checkColisionHorizontal(Board,[C1,C2])***: caso seja uma parede horizontal, verifica se já existe uma parede nessa posição;
- ***checkColisionHorizontalAux(Board,[P1,P2],Return)***: verifica se a colocação da nova parede vai cortar alguma parede vertical, de forma a impedir "cruzamentos".
- ***checkColisionVertical(Board,[C1,C2])***: caso seja uma parede vertical, verifica se já existe uma parede nessa posição:
- ***checkColisionVerticalAux(Board,[P1,P2],Return)***: verifica se a colocação da nova parede vai cortar alguma parede horizontal, de forma a impedir "cruzamentos".
- ***checkPawnsPath(Board)***: em ambos os casos, executa o predicado ***checkPath(Board,X,Y,[C1,C2],[C3,C4])*** para todos os peões em jogo, verificando se, após a colocação da nova parede, cada peão conseguiria chegar as duas posições iniciais adversárias.

Para se mover um peão utilizam-se os seguintes predicados de verificação:

- ***canPassWalls(Board,[P1,P2],Char)***: verifica se uma parede impede o movimento;
- ***checkPawnColision(NewX,NewY)***: verifica se na posição de destino, não se encontra nenhum peão;

2.4. Execução de Jogadas

Existem dois tipos de ações no jogo: mover um peão de posição ou colocar uma parede. O predicado para executar os movimentos dos peões é:

- ***move(Board, [P1,P2], Char, L1, NewBoard, PlayerChar)***: *Board* é o tabuleiro atual, *P1* e *P2* é a posição do peão que vai ser movido, *Char* é o tipo de movimento (‘NN’,’N’,’SS’ etc…) , *L1* é a nova posição do peão, *NewBoard* é o tabuleiro atualizado, *PlayerChar* é o caractere que representa cada jogador na matriz.

```
move(Board,[P1,P2],Char,L1,NewBoard,PlayerChar):-
    (Char=='NN',NewPosX is P1-2,NewX is 2*NewPosX-1,NewY is 2*P2-1,getElementFromMatrix(Board,NewX,NewY,1,1,Value),
        (Value\=p,Value\=j,replaceMatrix(Board,NewX,NewY,1,PlayerChar,StackBoard);StackBoard=Board),
        changePawn(NewPosX,P2,L1);
    Char=='N',NewPosX is P1-1,NewX is 2*NewPosX-1,NewY is 2*P2-1,getElementFromMatrix(Board,NewX,NewY,1,1,Value),
        (Value\=p,Value\=j,replaceMatrix(Board,NewX,NewY,1,PlayerChar,StackBoard);StackBoard=Board),
        changePawn(NewPosX,P2,L1);
    Char=='SS',NewPosX is P1+2,NewX is 2*NewPosX-1,NewY is 2*P2-1,getElementFromMatrix(Board,NewX,NewY,1,1,Value),
        (Value\=p,Value\=j,replaceMatrix(Board,NewX,NewY,1,PlayerChar,StackBoard);StackBoard=Board),
        changePawn(NewPosX,P2,L1);
    Char=='S',NewPosX is P1+1,NewX is 2*NewPosX-1,NewY is 2*P2-1,getElementFromMatrix(Board,NewX,NewY,1,1,Value),
        (Value\=p,Value\=j,replaceMatrix(Board,NewX,NewY,1,PlayerChar,StackBoard);StackBoard=Board),
        changePawn(NewPosX,P2,L1);
    Char=='OO',NewPosY is P2-2,NewX is 2*P1-1,NewY is 2*NewPosY-1,getElementFromMatrix(Board,NewX,NewY,1,1,Value),
        (Value\=p,Value\=j,replaceMatrix(Board,NewX,NewY,1,PlayerChar,StackBoard);StackBoard=Board),
        changePawn(P1,NewPosY,L1);
    Char=='O',NewPosY is P2-1,NewX is 2*P1-1,NewY is 2*NewPosY-1,getElementFromMatrix(Board,NewX,NewY,1,1,Value),
        (Value\=p,Value\=j,replaceMatrix(Board,NewX,NewY,1,PlayerChar,StackBoard);StackBoard=Board),
        changePawn(P1,NewPosY,L1);
    Char=='EE',NewPosY is P2+2,NewX is 2*P1-1,NewY is 2*NewPosY-1,getElementFromMatrix(Board,NewX,NewY,1,1,Value),
        (Value\=p,Value\=j,replaceMatrix(Board,NewX,NewY,1,PlayerChar,StackBoard);StackBoard=Board),
        changePawn(P1,NewPosY,L1);
    Char=='E',NewPosY is P2+1,NewX is 2*P1-1,NewY is 2*NewPosY-1,getElementFromMatrix(Board,NewX,NewY,1,1,Value),
        (Value\=p,Value\=j,replaceMatrix(Board,NewX,NewY,1,PlayerChar,StackBoard);StackBoard=Board),
        changePawn(P1,NewPosY,L1);
    Char=='SE',NewPosX is P1+1,NewPosY is P2+1,NewX is 2*NewPosX-1,NewY is 2*NewPosY-1,
        getElementFromMatrix(Board,NewX,NewY,1,1,Value),(Value\=p,Value\=j,
        replaceMatrix(Board,NewX,NewY,1,PlayerChar,StackBoard);StackBoard=Board),changePawn(NewPosX,NewPosY,L1);
    Char=='NE',NewPosX is P1-1,NewPosY is P2+1,NewX is 2*NewPosX-1,NewY is 2*NewPosY-1,
        getElementFromMatrix(Board,NewX,NewY,1,1,Value),(Value\=p,Value\=j,
        replaceMatrix(Board,NewX,NewY,1,PlayerChar,StackBoard);StackBoard=Board),changePawn(NewPosX,NewPosY,L1);
    Char=='NO',NewPosX is P1-1,NewPosY is P2-1,NewX is 2*NewPosX-1,NewY is 2*NewPosY-1,
        getElementFromMatrix(Board,NewX,NewY,1,1,Value),(Value\=p,Value\=j,
        replaceMatrix(Board,NewX,NewY,1,PlayerChar,StackBoard);StackBoard=Board),changePawn(NewPosX,NewPosY,L1);
    Char=='SO',NewPosX is P1+1,NewPosY is P2-1,NewX is 2*NewPosX-1,NewY is 2*NewPosY-1,
        getElementFromMatrix(Board,NewX,NewY,1,1,Value),(Value\=p,Value\=j,
        replaceMatrix(Board,NewX,NewY,1,PlayerChar,StackBoard);StackBoard=Board),changePawn(NewPosX,NewPosY,L1)),
    (LastX is 2*P1-1, LastY is 2*P2-1,getElementFromMatrix(Board,LastX,LastY,1,1,LastValue) ,LastValue\=p,
    LastValue\=j, replaceMatrix(StackBoard,LastX,LastY,1,c,NewBoard);NewBoard=StackBoard).
```

Figura 8 - Predicado para movimentação dos peões

O predicado para colocar paredes verticais e horizontais é:

- ***placeWall(Vert,Hor,Board,NewBoard,NewVert,NewHor)***: *Vert* é o número de paredes verticais que o jogador pode colocar, *Hor* é o número de paredes horizontais que o jogador pode colocar, *Board* é o tabuleiro do jogo, *NewBoard* é o tabuleiro atualizado, *NewVert* e *NewHor* é o numero atualizado de paredes verticais e horizontais com que o jogador ficou após a jogada.

```
placeWall(Vert,Hor,Board,NewBoard,NewVert,NewHor):-chooseWall(Vert, Hor, Choice,[X,Y]),
    ((Choice==1,Vert>0,checkColisionVertical(Board,[X,Y]),P1 is 2*X-1,P2 is 2*Y,
    replaceMatrix(Board,P1,P2,1,q,StackBoard),P3 is 2*X+1,replaceMatrix(StackBoard,P3,P2,1,q,NewBoard),
    checkPawnsPath(NewBoard),NewVert is Vert-1,NewHor is Hor;
    Choice==2,Hor>0,checkColisionHorizontal(Board,[X,Y]),P1 is 2*X,replaceMatrix(Board,P1,Y,1,w,StackBoard),
    P2 is Y+1,replaceMatrix(StackBoard,P1,P2,1,w,NewBoard),checkPawnsPath(NewBoard),
    NewVert is Vert,NewHor is Hor-1);placeWall(Vert,Hor,Board,NewBoard,NewVert,NewHor)).
```

Figura 9 - Predicados para colocação das paredes

2.5. Avaliação do Tabuleiro

A avaliação das diversas jogadas disponíveis em cada momento é feita essencialmente pelo seguinte predicado:

- ***availablePositions(Board,[P1,P2])***: recebe o tabuleiro e calcula todas as jogadas possíveis para um dado peão (P1 e P2); atualiza a lista numa variável dinâmica.

```
availablePositions(Board,[P1,P2]):-
canPassWalls(Board,[P1,P2],'NN'),NewPosX is P1-2,NewX is 2*NewPosX-1,NewY is 2*P2-1,
checkPawnColision(NewX,NewY),(\+ availableList(List),asserta(availableList(['NN'])));
availableList(List),\+ member('NN',List),retract(availableList(_)),append(['NN'],List,NewList),
asserta(availableList(NewList))),fail;
canPassWalls(Board,[P1,P2],'N'),NewPosX is P1-1,NewX is 2*NewPosX-1,NewY is 2*P2-1,
checkPawnColision(NewX,NewY),(\+availableList(List),asserta(availableList(['N'])));
availableList(List),\+ member('N',List),retract(availableList(_)),append(['N'],List,NewList),
asserta(availableList(NewList))),fail;
canPassWalls(Board,[P1,P2],'SS'),NewPosX is P1+2,NewX is 2*NewPosX-1,NewY is 2*P2-1,
checkPawnColision(NewX,NewY),(\+availableList(List),asserta(availableList(['SS'])));
availableList(List),\+ member('SS',List),retract(availableList(_)),append(['SS'],List,NewList),
asserta(availableList(NewList))),fail;
canPassWalls(Board,[P1,P2],'S'),NewPosX is P1+1,NewX is 2*NewPosX-1,NewY is 2*P2-1,
checkPawnColision(NewX,NewY),(\+availableList(List),asserta(availableList(['S'])));
availableList(List),\+ member('S',List),retract(availableList(_)),append(['S'],List,NewList),
asserta(availableList(NewList))),fail;
canPassWalls(Board,[P1,P2],'OO'),NewPosY is P2-2,NewX is 2*P1-1,NewY is 2*NewPosY-1,
checkPawnColision(NewX,NewY),(\+availableList(List),asserta(availableList(['OO'])));
availableList(List),\+ member('OO',List),retract(availableList(_)),append(['OO'],List,NewList),
asserta(availableList(NewList))),fail;
canPassWalls(Board,[P1,P2],'O'),NewPosY is P2-1,NewX is 2*P1-1,NewY is 2*NewPosY-1,
checkPawnColision(NewX,NewY),(\+availableList(List),asserta(availableList(['O'])));
availableList(List),\+ member('O',List),retract(availableList(_)),append(['O'],List,NewList),
asserta(availableList(NewList))),fail;
canPassWalls(Board,[P1,P2],'EE'),NewPosY is P2+2,NewX is 2*P1-1,NewY is 2*NewPosY-1,
checkPawnColision(NewX,NewY),(\+availableList(List),asserta(availableList(['EE'])));
availableList(List),\+ member('EE',List),retract(availableList(_)),append(['EE'],List,NewList),
asserta(availableList(NewList))),fail;
canPassWalls(Board,[P1,P2],'E'),NewPosY is P2+1,NewX is 2*P1-1,NewY is 2*NewPosY-1,
checkPawnColision(NewX,NewY),(\+availableList(List),asserta(availableList(['E'])));
availableList(List),\+ member('E',List),retract(availableList(_)),append(['E'],List,NewList),
asserta(availableList(NewList))),fail;
canPassWalls(Board,[P1,P2],'NO'),NewPosX is P1-1,NewX is 2*NewPosX-1,NewPosY is P2-1,NewY is 2*NewPosY-1,
checkPawnColision(NewX,NewY),(\+availableList(List),asserta(availableList(['NO'])));
availableList(List),\+ member('NO',List),retract(availableList(_)),append(['NO'],List,NewList),
asserta(availableList(NewList))),fail;
canPassWalls(Board,[P1,P2],'NE'),NewPosX is P1-1,NewX is 2*NewPosX-1,NewPosY is P2+1,NewY is 2*NewPosY-1,
checkPawnColision(NewX,NewY),(\+availableList(List),asserta(availableList(['NE'])));
availableList(List),\+ member('NE',List),retract(availableList(_)),append(['NE'],List,NewList),
asserta(availableList(NewList))),fail;
canPassWalls(Board,[P1,P2],'SE'),NewPosX is P1+1,NewX is 2*NewPosX-1,NewPosY is P2+1,NewY is 2*NewPosY-1,
checkPawnColision(NewX,NewY),(\+availableList(List),asserta(availableList(['SE'])));
availableList(List),\+ member('SE',List),retract(availableList(_)),append(['SE'],List,NewList),
asserta(availableList(NewList))),fail;
canPassWalls(Board,[P1,P2],'SO'),NewPosX is P1+1,NewX is 2*NewPosX-1,NewPosY is P2-1,NewY is 2*NewPosY-1,
checkPawnColision(NewX,NewY),(\+availableList(List),asserta(availableList(['SO'])));
availableList(List),\+ member('SO',List),retract(availableList(_)),append(['SO'],List,NewList),
asserta(availableList(NewList))),fail;true.
```

FIGURA 10 - DEFINIÇÃO DO PREDICADO AVAILABLEPOSITIONS

2.6. Final do Jogo

O fim do jogo é determinado pelo predicado a seguir descrito:

- ***checkWinners(End)***: verifica se o jogo terminou, retornando *End* a 1 caso os dois peões de um jogador estejam nas posições iniciais do adversário; caso contrario, retorna *End* a 0.

```
checkWinners(End):-pawn1([P1v,P1h]),pawn2([P2v,P2h]),pawn3([P3v,P3h]),pawn4([P4v,P4h]),
(P1v==11,P2v==11,(P1h==4;P1h==8),(P2h==4;P2h==8),nl,write('Player 1 wins'),nl,End is 1;
P3v==4,P4v==4,(P3h==4;P3h==8),(P4h==4;P4h==8),nl,write('Player 2 wins'),nl,End is 1);End is 0,true.
```

FIGURA 11 - DEFINIÇÃO DO PREDICADO CHECKWINNERS

2.7. Jogada do Computador

A jogada autónoma por parte da máquina é conseguida através dos seguintes predicados:

- ***movePawnRandomly(Board,Turn,NewBoard)***: recebe o tabuleiro atual do jogo e move um peão aleatoriamente, atualizando o tabuleiro em *NewBoard*;
- ***placeRandomWall(V,H,Board,NewBoard,NewVert,NewHor)***: *V* é o número de paredes verticais, *H* é o número de paredes horizontais, *Board* é o tabuleiro recebido; coloca aleatoriamente uma parede horizontal ou vertical, escolhendo da lista de posições válidas de paredes; atualiza a informação em *NewBoard*, N*ewVert* e N*ewHor*;
- ***smartMovementCPU(Board,Turn,NewBoard)***: usa um algoritmo de *flood fill* para calcular o caminho mais curto dos peões à posição final mais próxima dos mesmos, move-os nesse sentido e atualiza o tabuleiro;
- ***smartWallCPU(Board,Turn,NewBoard,Vert,Hor,NewVert,NewHor)***: ordena a lista de um tipo de paredes possíveis (horizontais ou verticais) de acordo com a proximidade aos 2 peões adversários; escolhe o primeiro elemento da lista e coloca a parede; atualiza a informação no fim;

3. Interface com o Utilizador

Quando o programa é iniciado surge o seguinte ecrã:

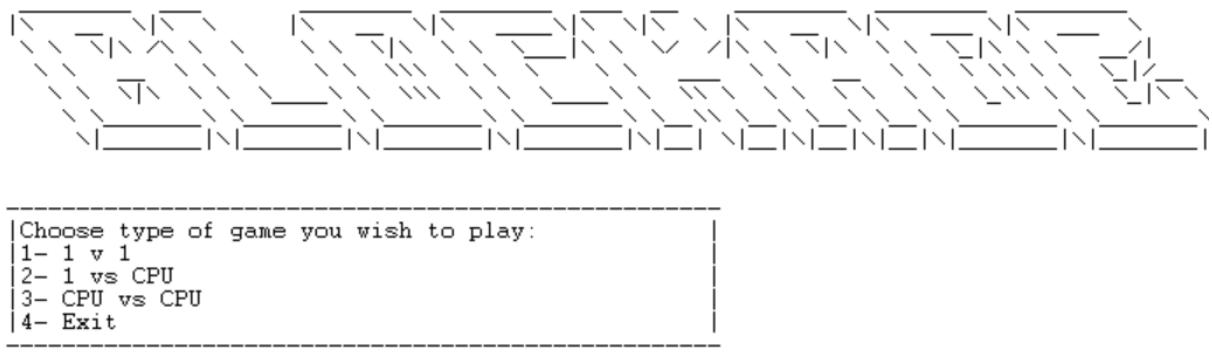


FIGURA 12 - ECRÃ INICIAL

Aqui podemos ver o menu principal em que surgem as opções de jogar *player* contra *player*, *player* contra máquina, máquina contra máquina e sair do programa. O jogador deve escolher o número da opção que deseja seguido de um ponto final. Este procedimento deve ser usado em todos os menus que aparecerão no decorrer do jogo.

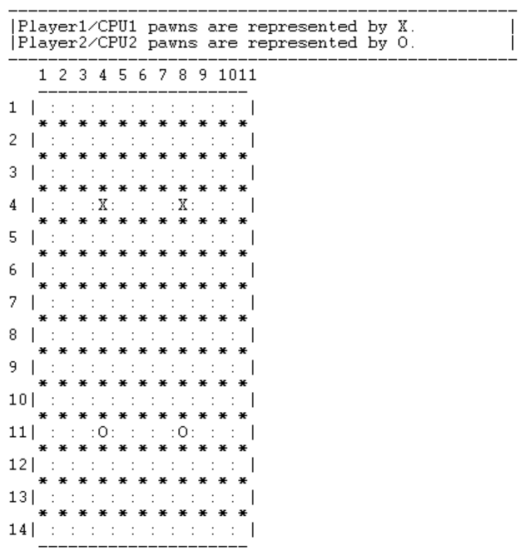


FIGURA 13 - INICIO DO JOGO

Aqui podemos observar um ecrã de inicio do jogo. A mensagem no topo indica-nos a que jogador ou máquina (dependendo do modo de jogo) pertencem os peões. Imediatamente abaixo temos o tabuleiro pronto para começar o jogo. A seguir a isto aparece o seguinte menu (exceto no modo em que só jogam máquinas):

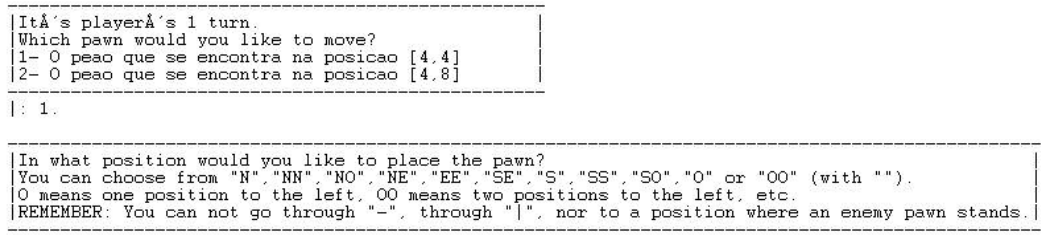


FIGURA 14 - MOVIMENTÇÃO DE PEÕES

Neste menu é pedido que o jogador escolha o peão que quer mover. De seguida tem de ser escolhida a direção na qual o peão vai ser movido. Imediatamente a seguir surgirá o menu para colocação da parede (caso o jogador ou máquina ainda tenha paredes para colocar e caso o modo de jogo não seja máquina contra máquina).

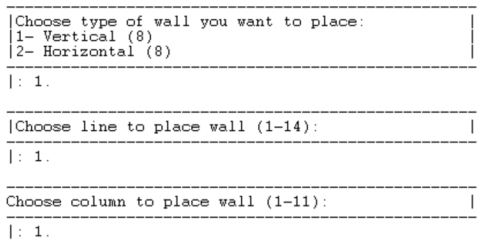


FIGURA 15 - COLOCAÇÃO DE PAREDES

Aqui o jogador terá de escolher se quer colocar uma parede vertical ou horizontal e a posição onde a quer colocar (linha e coluna). O processo de movimentação de peões e colocação de paredes vai sendo repetido (automaticamente no caso máquina contra máquina e através dos menus apresentados acima no caso em que estejam jogadores presentes) até ser encontrado um vencedor. Quando isto acontece surge uma mensagem e informar quem foi o vencedor e o programa retorna ao menu inicial.

Conclusões

Durante a realização deste trabalho podemos aprofundar os conhecimentos adquiridos durante as aulas relativamente à programação lógica.

Concluímos que cumprimos os objetivos propostos pelos docentes, sendo que poderíamos aperfeiçoar alguns aspetos do trabalho se o tempo assim o permitisse, sendo estes: transformar predicados grandes em predicados menores, mudar a forma como tratamos os peões e tabuleiro e melhorar a colocação de paredes por parte do CPU no modo inteligente.

Bibliografia

https://en.wikipedia.org/wiki/Blockade_(board_game)

http://boardgamegeek.com/boardgame/2559/blockade

Anexos

O código fonte encontra-se na paste *source* anexada a este relatório.