BLOCKADE

Relatório Final

Mestrado Integrado em Engenharia Informática e Computação

Programação em Lógica

Grupo Blockade_1:

Mário Fernandes – up201201705

Luís Alves – up201405308

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

13 de novembro de 2016

Resumo

O trabalho descrito neste relatório trata-se da implementação do jogo de tabuleiro Blockade na linguagem PROLOG.

Durante a execução foi entendido grande parte da utilidade geral da linguagem PROLOG, sendo esta bastante diferente da maior parte das linguagens aprendidas até hoje, sendo que é uma linguagem funcional. Este tipo de linguagens trata os problemas apresentados como uma série de avaliações de predicados que constam na sua base de dados, que por sua vez é implementada pelo programador. Dito isto, o grupo adotou um método de trabalho em que qualquer predicado implementado era testado de seguida com o auxílio do debug que o programa SICSTUS proporciona.

Com a realização deste trabalho foi visto que PROLOG não necessita de muitas linhas de código para executar um comando, mas comparando a linguagens como C ou JAVA, por vezes a compreensão de código poderá não ser a mais facil, visto que os programas são feitos à base de recursividade.

Contents

1. Introdução

No âmbito da cadeira de Programação em Lógica foi consensual no grupo realizar um trabalho sobre o jogo de tabuleiro Blockade uma vez que ambos apreciamos a essência e mecânicas do jogo. Considerando que o maior objetivo do trabalho é a aplicação da linguagem PROLOG para resolver problemas reais, decidimos adoptar algumas metodologias que nos ajudassem mais na aprendizagem desta mesma, tal como o uso da recursividade em grande escala.

2. O Jogo Blockade

Blockade é um jogo de estratégia para dois jogadores inventado por Mirko Marchesi e Philip Slater e publicado por várias empresas ao longo do tempo sendo que a primeira foi a Lakeside Industries, em 1975.

O termo Blockade (ou em português, bloqueio), vai ao encontro da essência do jogo e sugere um cenário de guerra passiva-agressiva que tem como objetivo chegar a uma base do oponente antes que o adversário o consiga fazer num tabuleiro 11*14 com células quadradas. Dito isto, cada jogador é munido com duas peças principais (peões) e com dezoito paredes: nove verdes e nove azuis. As paredes verdes e azuis apenas se podem colocar verticalmente e horizontalmente, respetivamente, ocupando duas células cada uma.

O jogo desenrola-se com cada jogador jogando à vez e cada jogada consiste em mexer apenas um dos seus peões para uma casa adjacente uma ou duas vezes; após esta fase é dada a escolha ao jogador de colocar uma parede no tabuleiro ou preservá-las para jogadas seguintes. Terá sempre de existir um caminho possível entre todos os peões e ambas as casas objetivo, ou seja, é proíbido jogadas como colocar 4 paredes à volta de uma base.

O jogo termina quando um dos jogadores mover um dos seus peões para uma das bases do oponente, sendo esse o vencedor.

3. Lógica do Jogo

3.1. Representação do Estado do Jogo

Existem 3 estados de jogo distintos: estado inicial, estado em que o jogador 1 ganha e estado em que o jogador 2 ganha. O que importa reter destes estados em termos de código é o seguinte:

- Estado inicial: Estão os 4 peões nas suas bases, sendo estas para o jogador 1 e 2 as casas com coordenadas de [4,4] , [8,4] e [4,11] , [8,11] respetivamente. Não existem paredes no tabuleiro.

- Estado final em que o jogador 1 ganha: um dos peões do jogador 1 está numa das bases do oponente, [4,11] ou [8,11].

- Estado final em que o jogador 2 ganha: um dos peões do jogador 2 está numa das bases do oponente, [4,4] ou [8,4].

Existem ainda alguns estados de jogo intermédios que não têm uma representação específica, como por exemplo todas as fases do jogo depois de alguém fazer uma jogada ou a 2ª parte da jogada de um jogador (o posicionamento da parede). Ora, como em termos de código não seria eficaz representar todas estas fases intermédias, apenas teremos declarados os 3 estados acima referidos.

As posições das peças são representadas com uma lista das várias linhas do tabuleiro, sendo estas, também listas, e assim podemos identificar todas as células do tabuleiro. Nestas células estão presentes variáveis numéricas (ou '_' para representar qualquer variável) que no desenrolar do jogo são substituídas por espaços vazios, 'A' e 'B' para o jogador 1 e 'C' e 'D' para o jogador 2. As células contêm variáveis pois com o decorrer do jogo estas poderão ser alteradas.

No código final, a representação do estado de jogo final foi alterada para apenas uma função que verifica se as bases estão com alguma peça contrária, sendo assim as figuras representadas por apenas algumas linhas de código.

As paredes são também armazenadas numa lista de linhas, sendo que as paredes verticais possuem menos um elemento em cada linha em relação ao estado das peças e as paredes

horizontais menos uma linha. A razão para tal deve-se a facto de as paredes estarem posicionadas não em células do tabuleiro, mas nos espaços entre estas. '0' representa um local onde pode ser colocada uma parede, enquanto que o '1' representa um local onde já existe uma parede.

Como tal, a cada iteração do jogo, existem 3 listas de listas, ou matrizes, para representar o seu estado: uma para a posição das peças, uma para as paredes verticais, e uma para as horizontais.

3.2. Visualização do Tabuleiro

A visualização do tabuleiro de jogo é feita com o predicado display_board que recebe as três matrizes necessárias para a representação do jogo como já acima referido. As matrizes são listas de listas, e o predicado vai processando cada linha individualmente fazendo o display_line. Este display_line escreve no programa todas as células da lista recebida, intercaladas ou não por uma parede (dependo da existência de uma parede nessa posição específica). Existe ainda um predicado específico para tratar do caso em que a lista fica vazia (cujo objetivo é simplesmente mudar de linha).

Entre cada linha existe a possibilidade de escrever paredes horizontais de modo análogo.

Após o processamento de uma linha, o predicado display_board_m irá ser chamado outra vez (existindo assim uma recursividade) e fará todo o processamento idêntico ao anterior, mas agora com a próxima linha. Quando as matrizes 'B' e 'V' ficarem vazias (têm garantidamente o mesmo comprimento), existe um predicado encarregue de tratar deste caso (que apenas escreve uma linha por motivos estéticos), acabando assim, o processo de visualização do tabuleiro.

Durante este trabalho encontrámos uma pequena dificuldade na representação do tabuleiro pois tentámos utilizar caracteres especiais mas rapidamente percebemos que dependendo do computador e tipo de letra utilizado, a visualização poderá ficar desformatada. Qualquer tipo de letra em que todos os caracteres têm a mesma largura deverá funcionar corretamente, pelo que aconselhamos a utilização do "Courier New" normal, tamanho 12.

3.3. Lista de Jogadas Válidas

Para detectar se uma jogada ou input do utilizador é válido foi utilizado um sistema de predicados inicilizados com “check”. Estes predicados apresentam uma “afirmação” em prolog que se estiver verdadeira, o predicado que os chamou avança, se não entra num ciclo até o utilizador fazer uma jogada válida.

Um dos cuidados que foi tido com a implementação destes predicados foi o facto destes não fazerem alterações no tabuleiro atual, apenas verificarem se certas células estão vazias. O mesmo se verifica nos predicados de verificação de posicionamento das paredes.

O predicado essencial neste tópico foi o “getCell(Board,Row,Column,Piece) :- nth0(Row, Board, NewRow), nth0(Column, NewRow, Piece).” que foi feito com o objetivo de pesquisar numa lista de listas o posicionamento de uma peça. Colocando as facilidades que PROLOG nos oferece, também foi utilizado para a pesquisa de coordenadas de uma dada peça.

Predicados deste tópico:

checkDestination(+Row,+Column,+(direcao*),+Board,-NewRow,-NewColumn,+V,+H), para a verificação de um movimento;

checkBoard(+Row,+Column,+(direção*),+Board,-NewRow,-NewColumn,+V,+H), para a verificação de poder saltar sobre um oponente;

checkSetWall(+(direção *),+V,+H,+Row,+Column), para a verificação de poder colocar uma parede.

Nota: “direção *” significa um input do utilizador previamente verificado como por exemplo “w”, H significa matriz de paredes horizontais, V significa matriz de paredes verticais.

3.4. Execução de Jogadas

A execução de movimentos é realizado com o predicado move(y,+Piece,+Direction,+Board,-NBoard,+V,+H,-Answer), que retorna Board modificado sob a forma de NBoard. Nesta função existem ainda os parâmetros "y" que indica que é uma jogada que o utilizador deseja fazer, "V" e "H" que são as listas de paredes verticais e horizontais e "Answer" que é um indicador de que o jogo acabou ou ainda está em curso. Este predicado para além de chamar os predicados "check" também faz o display do tabuleiro.

Todas as modificações são feitas via predicados inicializados por "set", mais propriamente o predicado setCell(Board, Row, Column, Object, NewBoard), bastante genérico, que altera uma célula numa lista de listas. Este predicado chama duas vezes o predicado setPart(Pos, Object, List, NewList):-length(TempList,Pos),append(TempList,[_|Y],List), append(TempList,[Object|Y],NewList)., que, utilizando appends, começa por receber a linha em questão e alterando a célula dessa mesma; na segunda vez que é chamado, o setPart tem como objetivo colocar a nova linha criada no tabuleiro.

Finalmente a jogada de colocação de uma parede é feita com o predicado setWalls(y,+V,+H,-NV,-NH,+HwallsCount,+VwallsCount,-CountH,-CountV) em que V, H, NV e NH são as matrizes velhas e novas de paredes e os outros 4 seguintes argumentos são os contadores de paredes (para prevenir que alguem com 0 paredes consiga colocar alguma parede desse tipo).

3.5. Avaliação do Tabuleiro

A avaliação do estado do jogo é feita nos predicados "game" "gamevscomputer" e "gameauto" para jogos de humanos contra humanos, humano contra computador e computador contra computador, respetivamente. Todos estes predicados têm um método idêntico: chamam os diferentes predicados que tratam das jogadas de um jogador especifico e, no final, chamam de novo o próprio predicado, avaliando a cada movimento de peão se se está perante um jogo acabado.

Os predicados de jogadas foi inicialmente feito com uma única função "play" mas rápidamente se alterou para "player1Move" e "player2Move" para tornar mais clara a implementação e para facilitar bastante a detecção de erros que surgiam à medida que se

desenvolvia o jogo. A separação dos predicados também ajudou nos movimentos específicos de cada jogador e que cada um pode fazer, por exemplo, no peão de escolha do jogador.

Estes predicados são constituídos por 3 fases, a primeira jogada em que o jogador escolhe o peão que quiser descolar e a sua direção, a segunda em que pode escolher se deseja deslocar o mesmo peão mais alguma vez e em que direção e a terceira que é a colocação das paredes. Em todas estas fases são verificadas as condições de jogadas válidas e é adaptada a jogada do utilizador, por exemplo, no caso de estar um peão inimigo no caminho. Neste último caso o peão do jogador irá saltar por cima desse peão inimigo.

Predicado deste tópico:

player1Move(+Board,+V,+H,-NBoard,-NV,-NH,+HwallsCount,+VwallsCount,-CountH, -CountV, -Over).

3.6. Final do Jogo

A verificação de que o jogo acabou é feita com um predicado do tipo “check” chamado checkGameOver(+Board,-Over). Este predicado está instanciado 9 vezes, verificando as 4 posições iniciais dos peões no tabuleiro se contêm um peão adversário. Se percorrer encontrar algum destes casos iguala a variavel Over a 1, indicando que o jogo acabou e imprime uma mensagem a constatar a equipa vencedora. Se percorrer os 8 predicados que assumem que o jogo acabou, entra no último em que a única função do predicado é igualar Over a 0, indicando que o jogo ainda está em curso. A verificação deste predicado é feito outra vez com auxílio do “getCell” e é feita sempre que existe um movimento válido de um peão.

3.7. Jogada do Computador

A inteligência artificial do jogo está implementada em 2 niveis: o nivel 0 em que o computador faz as escolhas todas ao acaso e o nivel 1 em que o computador tenta seguir sempre em frente com o peão mais próximo da casa objetivo e se não conseguir ou joga com o outro peão ou tenta seguir um caminho mais longe das paredes laterais. Todos os

movimentos por parte do computador estão no ficheiro “Blockade_ai.pl”, na qual o predicado principal é o “compMove”.

A versatilidade de todos os outros predicados feitos permitiu que a única mudança no ciclo de jogada fosse a obtenção de informação (feita automaticamente e não com “read”), existindo assim os predicados “gamevscomputer” e “gameauto” muito parecidos com o predicado de jogo entre humanos “game”. Uma outra alteração feita nestes ciclos foi a inclusão da dificuldade de cada computador (passada por argumento desde o início do jogo), que é utilizada na deteção da próxima jogada.

Predicados para este tópico:

compMove(+N,+Difficulty,+Board,+V,+H,-NBoard,-NV,-NH,+HwallsCount,+VwallsCount,-CountH,-CountV,-Over), no qual N é um contador para a escolha da peça;

getInfoComputer(+N,+Difficulty,+Board,-Piece,-Direction).

4. Interface com o Utilizador

A comunicação com o utilizador é feita via teclado, feita a partir de reads, o que significa que cada jogador terá de colocar um “.” no final do comando. Esta abordagem não é a melhor devido à necessidade do ponto final no fim de cada comando, devendo ter sido feita com recurso ao comando get_char; tal não aconteceu pois a utilização de get_char não esvazia o buffer do input stream após uma escrita do utilizador, provocando erros na próxima leitura.

O jogo é iniciado com o comando “newgame.” no qual é apresentado um menu ao utilizador para escolher um modo de jogo.

Conclusão

PROLOG é uma linguagem extremamente útil quando se necessita de implementar algo que exiga muita recursividade, como por exemplo, um jogo que estará sempre a executar os mesmos predicados ou calculos de possíveis jogadas nesse mesmo jogo. Com a execução deste projeto, ficou bastante mais clara a utilidade da linguagem face a outras não funcionais.

Relativamente ao nosso projeto, e na nossa opinião, teria facilitado bastante o uso de predicados como “assert” e “retract” e até mesmo “repeat” pois facilitava bastante o uso de recursividade. No entanto, o grupo achou melhor não utilizar estas funcionalidades para obter um melhor conhecimento da linguagem e da maneira como esta interpreta os comandos dados (como por exemplo, o “backtracking”).

Por fim podemos dizer que gostamos bastante de trabalhar neste projeto pois não só melhoramos o nosso conhecimento sobre um tipo de linguagem bastante diferente do habitual, bem como conseguimos programar de raiz um jogo interessante.

Bibliografia

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

https://www.harding.edu/fmccown/classes/comp440-f07/gameboard.html

Anexo A – Figuras sobre o jogo

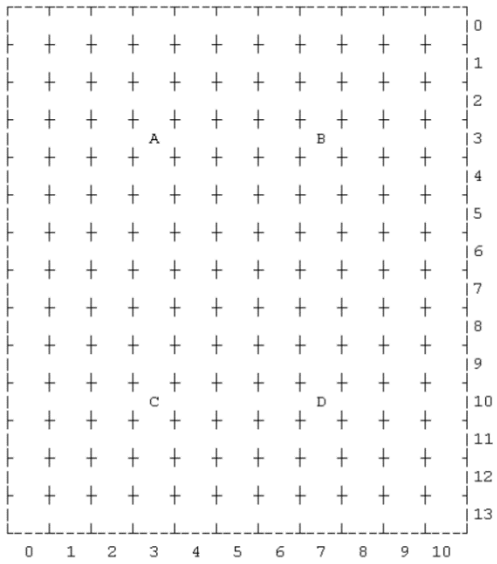

Fig 1 – Estado inicial do jogo

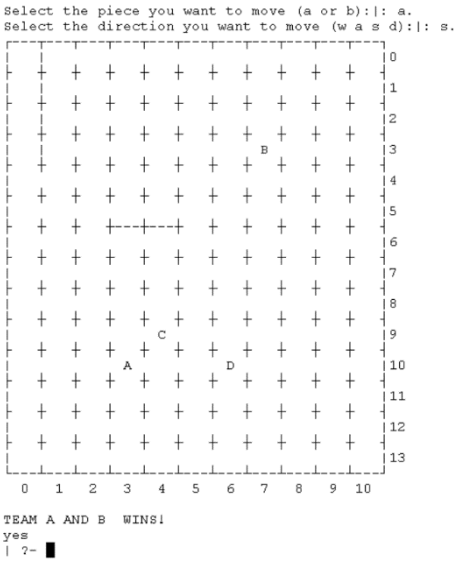

Fig 2 – Estado final do jogo

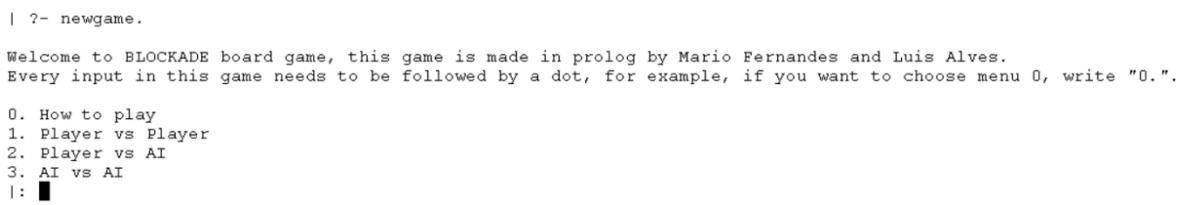

Fig 3 – Menu Inicial do jogo

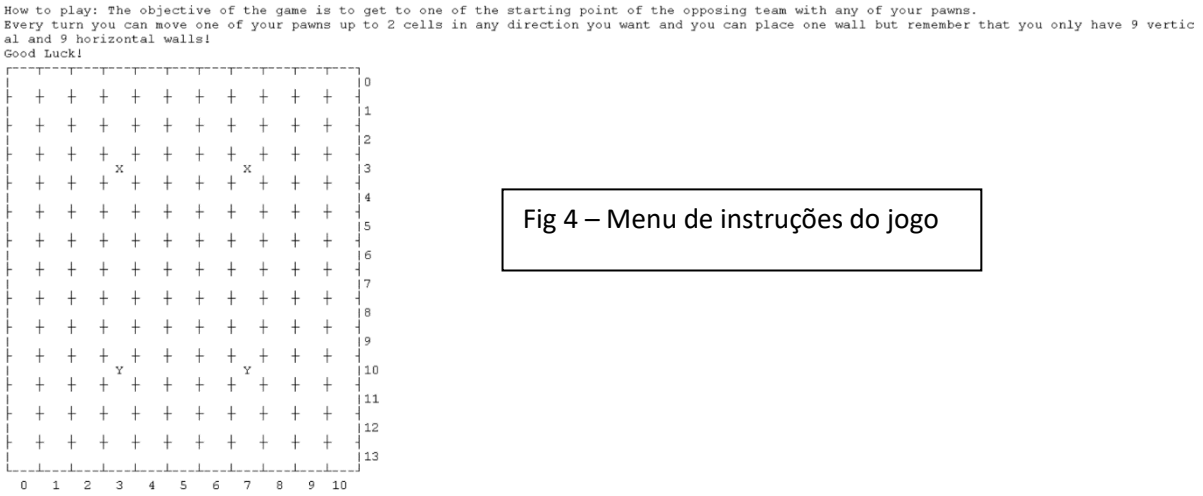

Fig 4 – Menu de instruções do jogo

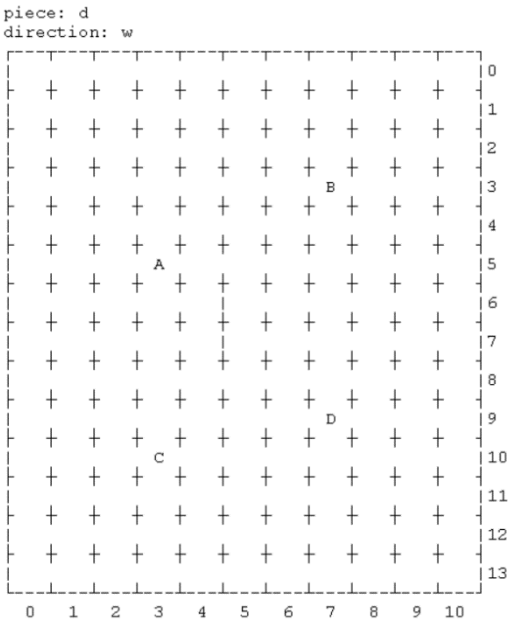

Fig 5 – Um movimento do computador

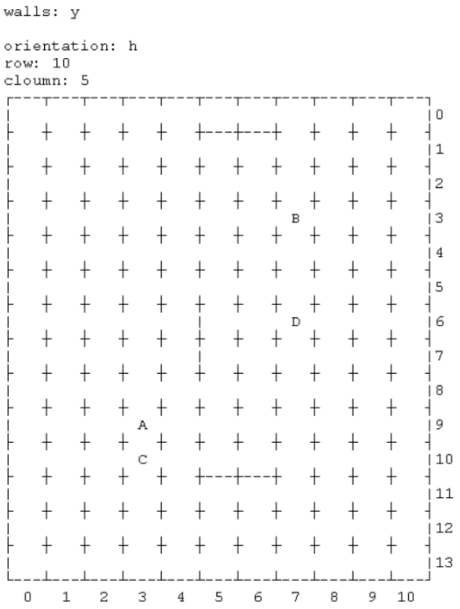

Fig 6 – Um posicionamento de paredes pelo computador