Faculdade de Engenharia da Universidade do Porto

MIEIC

Programação Lógica

Neutreeko

Grupo 3:
Duarte Frazão, 201605658
Pedro Costa, 201605339

November 18, 2018

1 Introdução

1.1 História

Neutreeko é um jogo de tabuleiro criado por Jan Kristian Haugland em 2001. É baseado em dois jogos outros jogos de tabuleiro (até o nome do jogo é uma ”mistura” de ambos):

- Neutron
- Teeko

1.2 Objectivo

Inicialmente, cada jogador tem um conjunto de 3 peças (quadrados e círculos). O tabuleiro inicial é o seguinte:

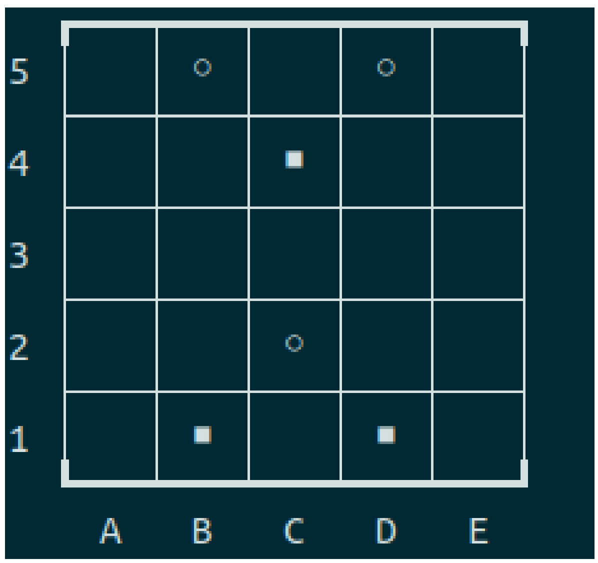

Figure 1: Tabuleiro Inicial

O objetivo é obter 3 em linha, diagonalmente ou ortogonalmente.

1.3 Regras

Os quadrados mexem-se primeiro. As peças podem deslizar em qualquer **posição ortogonal ou diagonal** à sua posição, sendo paradas por um campo ocupado ou pela fronteira do tabuleiro. No entanto, se o espaço imediatamente a seguir na direção do movimento já estiver ocupado, é considerado um movimento inválid; ou seja, tem de se mover sempre no mínimo um espaço livre. O objetivo é obter **3 em linha**, diagonalmente ou ortogonalmente.

Um estado intermédio do jogo pode, então, ser:

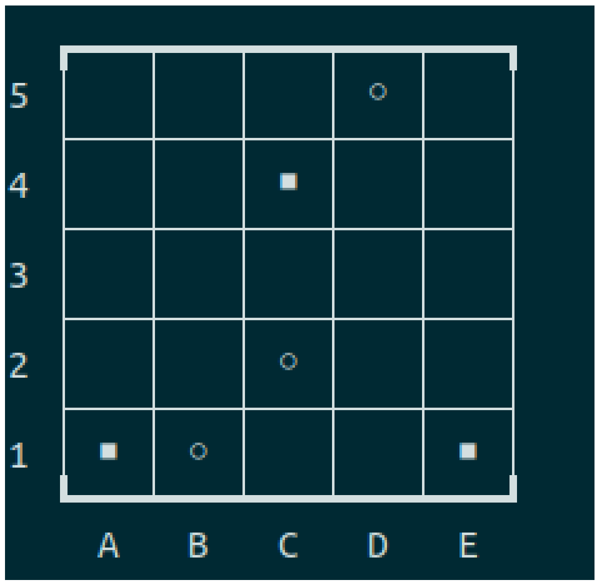

Figure 2: Tabuleiro Intermediário

Ocorre um **empate** quando a mesma disposição do tabuleiro ocorre 3 vezes.

2 Lógica do Jogo

2.1 Representação do Jogo

Internamente, o tabuleiro é representado por uma lista de listas, sendo cada elemento das listas internas uma célula, contento um átomo. O número 0 foi usado para a célula vazia, o número 1 para os círculos e o número 2 para os quadrados.

O Jogador é representado por uma variável, que pode ter os valores 0 ou 1. O Jogador 0 joga com os círculos (internamente 1), o Jogador 1 joga com os quadrados (internamente 0). Assim, para verificar se uma peça que um Jogador pretende mover, é apenas necessário ver se o valor dessa peça corresponde ao valor interno do Jogador mais 1.

Representação do estado inicial:

```
[[0, 1, 0, 1, 0],
[0, 0, 2, 0, 0],
[0, 0, 0, 0, 0],
[0, 0, 1, 0, 0],
[0, 2, 0, 2, 0]]
```

Representação do estado intermédio:

```
[[0, 0, 0, 1, 0],
[0, 0, 2, 0, 0],
[0, 0, 0, 0, 0],
[0, 0, 1, 0, 0],
[2, 1, 0, 0, 2]]
```

Representação do estado final:

```
[[0, 0, 0, 0, 0],
[0, 0, 0, 0, 2],
[0, 0, 0, 0, 0],
[0, 0, 0, 0, 0],
[2, 1, 1, 1, 2]]
```

2.2 Visualização do Jogo

O predicado display do tabuleiro é essencialmente recursiva. Percorre cada elemento da lista, linha a linha, célula a célula. Ao encontrar uma célula, traduz o seu conteúdo (0, 1 ou 2) para um caracter de Unicode que é depois colocado na consola. Para um aspeto mais esteticamente aprazível do tabuleiro, antes e depois de cada linha do tabuleiro, são introduzidas divisões (horizontais e verticais).

O output deste predicado foi já vista nas duas figuras anteriores. Uma outra situação, neste caso de vitória dos círculos, pode ser vista aqui:

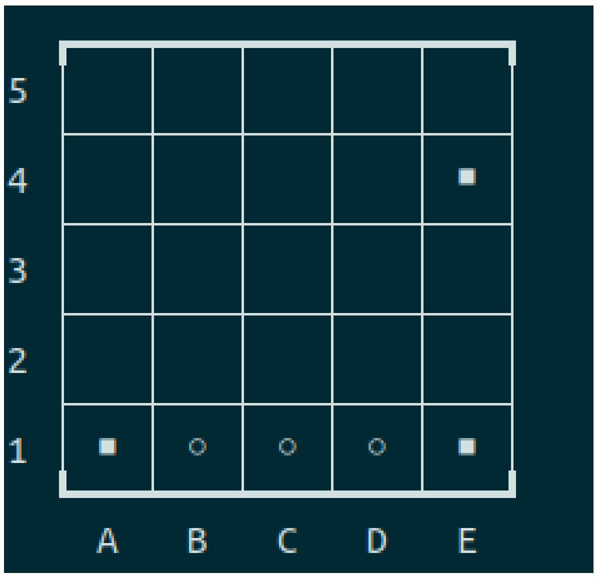

Figure 3: Tabuleiro Final

2.3 Lista de Jogadas Válidas

Dadas as regras do movimento expostas na secção das Regras, foi criado o seguinte predicado que valida um movimento sobre uma peça.

```
1  valid_move(Board,Player,Row,Col,Move, Piece):-
2      integer(Move),
3      between(1, 9, Move),
4      Move \= 5,
5      Piece =:= (Player + 1),
6      colTranslate(Move, MoveCol),
7      rowTranslate(Move, MoveRow),
8      NextRow is Row + MoveRow,
9      NextCol is Col + MoveCol,
10     length(Board, BoardLength),
11     checkBoundaries(NextRow, BoardLength),
12     checkBoundaries(NextCol, BoardLength),
13     checkEmpty(Board,NextRow,NextCol).
```

A direção do movimento, indicada por *Move* nos argumentos, é um valor inteiro, de 0 a 9, seguindo a direção tipica do teclado numérico. Dado que a peça tem de se movimentar sempre em alguma direção, o valor 5, central no teclado, não pode ser considerado (verificações efetuadas entre as linhas 2 e 4).

A seguir, verifica-se se a peça pertence efetivamente ao Jogador que fez o movimento, da forma apresentada na secção Representação do Jogo.

Feito isto, basta apenas verificar se o próximo espaço na direção do movimento está livre. Para isto, foram criadas duas funções, *colTranslate* e *rowTranslate*, que transformam um movimento em dois valores que representam, respetivamente, a deslocação provocada pelo movimento nas colunas e nas linhas. Por exemplo, um movimento de valor 7, localizado no canto superior esquerdo do teclado numérico, causaria um movimento vertical para a esquerda. Na nossa representação matricial do tabuleiro, isso implica recuar nas linhas e nas colunas, ficando tanto o *MoveRow* e *MoveCol* com -1. Tendo a posição do próximo espaço é só verificar se se encontra vazio, com o valor 0 na matriz.

Tendo o predicado que valida um movimento é trivial gerar todos os movimentos válidos, recorrendo do predicado *findall* do Prolog:

```
valid_moves(Board, Player, ListOfMoves):-
    NPiece is Player + 1,
    findall([Row, Col, Move],
            (
            getPiece(Row, Col, Board, NPiece),
            valid_move(Board, Player, Row, Col, Move, NPiece)
            ),
        ListOfMoves
        ).
```

Como o nosso predicado de validação já recebe uma peça, que o jogador escolhe previamente, é só necessário adicionar o passo extra de escolher uma peça do tabuleiro, através do predicado *getPiece*. Com o mecanismo de backtrack do Prolog e o predicado *findall* é possível encontrar as posições onde as peças do Jogador estão no Tabuleiro. Estas novas posições, agora instanciadas, são usadas no predicado valid_move e no final teremos uma lista, *MovesList*, onde cada elemento segue a estrutura [**Row, Col, Move**].

2.4 Execução de Jogadas

O predicado para efetuar um movimento é o seguinte:

```
move(Move, Piece, Board, NewBoard):-
        Move = [Row, Col, Dir],
        colTranslate(Dir, MoveCol),
        rowTranslate(Dir, MoveRow),
        replaceElemMatrix(Row, Col, 0, Board, NewBoard),
        length(Board, N),
        movePiece(Row, Col, MoveCol, MoveRow, Piece,
                    NBoard, OutBoard, N).
```

Traduz o movimento nas deslocações em linha e coluna respetivas e troca o elemento do tabuleiro onde está a peça que vai ser movida pelo valor 0, que representa um espaço livre. De seguida, chama um predicado auxiliar que efetua o movimento em si.

```
movePiece(Row, Col, MoveCol, MoveRow, Piece,
            Board, OutBoard, BoardLength):-
        NextRow is Row + MoveRow,
        NextCol is Col + MoveCol,
        checkBoundaries(NextRow, BoardLength),
        checkBoundaries(NextCol, BoardLength),
        getPiece(NextRow, NextCol, Board, NextPiece),
        NextPiece = 0,
        movePiece(NextRow, NextCol, MoveCol, MoveRow,
                Piece, Board, OutBoard, BoardLength);
        replaceElemMatrix(Row, Col, Piece, Board, OutBoard).
```

Aqui, é calculada a posição do próxima posição, somando o valor de desloção às coordenadas atuais, e obtém-se o valor da peça nessa posição. Se o valor for 0, espaço vazio, a peça pode continuar a mover-se; é efetuada uma chamada recursiva com as novas posições da peça. Se estiver ocupada, não se pode mover mais e o valor do tabuleiro nessa posição é alterado para o valor da peça, através do predicado *replaceElemMatrix*.

2.5 Final do Jogo

O predicado para detetar o fim do jogo é o seguinte:

```
game_over(Board, Player, Length):-
        K is Length - 1,
        (
                checkColumns(Board, Player, K);
                checkRows(Board, Player, K);
                checkBiggerDiagonals(Board, Length, Player)
        ).
```

Tenta detetar a ocurrência de final de jogo (três em linha) nas três formas possíveis: horizontalmente, verticalmente ou diagonalmente.

Para detetar empate temos o seguinte predicado:

```
checkDraw(Tab):-
        flatten2(Tab, List),
        (
                retract( map(List,N) ),
                NewN is N + 1,
                assertz( map(List, NewN) );
                NewN is 1,
                assertz(map(List, NewN))
        ),
        NewN > 2.
```

O anterior começa por colapsar o tabuleiro, que é representado como uma lista de listas, em apenas uma lista para facilitar o passo seguinte. De seguida, tenta remover uma clausula da base de dados igual ao tabuleiro atual, se existir é retirado e incrementasse o número de vezes que já apareceu o tabuleiro adicionando a cláusula atualizada, senão adiciona-se uma cláusula com esse tabuleiro (colapsado) e com o número de vezes que ocorreu a 1. No final verifica se o mesmo tabuleiro já aconteceu mais de duas vezes, caso onde é considerado um empate.

2.6 Avaliação do Tabuleiro

O predicado que avalia um tabuleiro é o seguinte:

```
value(Tab, Player, Value, Length):-
        game_over(Tab, Player, Length),Value = 1000;
        nextPlayer(Player, NextPlayer),
        game_over(Tab, NextPlayer, Length),Value = -1000;
        (
        checkDraw(Tab), !,

           checkDraw(Tab), !,
            resetMap(Tab), value = 0

           ;

           resetMap(Tab), fail
        );
        checkTwoConnected(Tab, Player, Length, Total),
        Total15 is Total * 15,
        valid_moves(Tab, Player, MovesList),
        nextPlayer(Player, NextPlayer),
        valid_moves(Tab, NextPlayer, EnemyMoves),
        length(MovesList, N),
        length(EnemyMoves, E),
        Value is Total15 + N - E.
```

Começa por testar se a jogada resulta numa vitória, sendo que nesse caso lhe dá um valor maior do que outro que possa existir. Se não for esse o caso, conta as vezes em que duas peças do jogador estão na mesma linha e juntas, em qualquer direção, e somando ao valor da jogada 15 por cada vez que isto se verifica.

Resta testar se ocorre um empate. Aqui o valor do tabuleiro é 0. Note-se que a nossa função de testar o empate modifica a base de dados interna relativamente ao predicado map. Como estamos só a testar possíveis movimentos é necessário após o teste remover a nova ocorrência do Tabuleiro (predicado *resetMap(Tab)*).

Se nada do anterior se verificar, soma o número de jogadas possíveis que pode fazer e subtrai ao número de jogadas possíveis que o adversário pode fazer. Com esta última fase de avaliação conseguímos prever e evitar que as peças do jogador fiquem bloqueadas e o contrário para o adversário.

2.7 Jogada do Computador

O predicado de escolha da jogada do computador é o seguinte:

```
choose_move(Board, Player, Level, Move):-
        (
        Level = 1,
                valid_moves(Board, Player, MovesList),
                length(MovesList, L),
                random(0, L, Index),
                nth0(Index, MovesList, Move)
                ;
        Level = 2,
                move_and_evaluate(Board, Player, Move)
        ).
```

Tem dois níveis. O primeiro encontra todas as jogadas válidas para o jogador e escolha uma de forma aleatória. O segundo escolhe a jogada com uma abordagem gananciosa usando o predicado:

```
move_and_evaluate(+Board, +Player, -BestMove).
```

Este predicado avalia recorrendo ao predicado anterior

```
value(+Tab, +Player, -Value, +Length).
```

e determina todas as jogadas possíveis escolhendo a melhor relativamente à avaliação feita.

2.8 Minimax

Para conseguir um adversário mais inteligente tentámos implementar o algoritmo Minimax. O algoritmo permite para uma profundidade especificada prever o estado do jogo através da expansão da árvore de estados futuros possíveis.

Em seguida apresentamos um esquema para ilustrar o algoritmo:

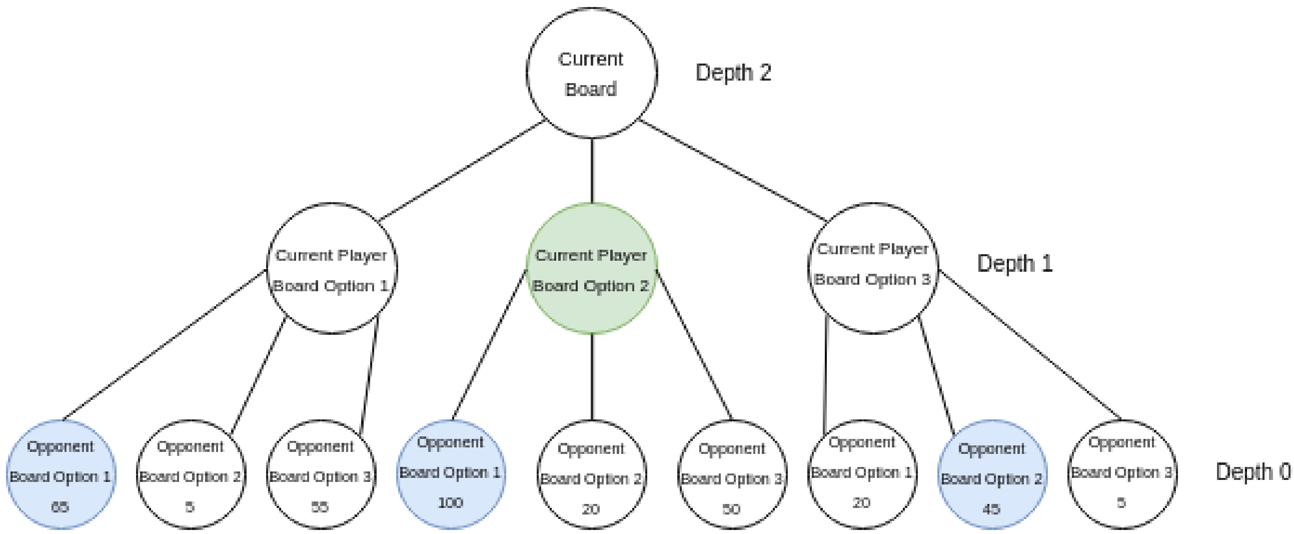

Figure 4: Esquema do algoritmo Minimax

O predicado usado para executar o algoritmo é o seguinte:

```
minimax(Tab, Player, State, Depth, NextVal, NextTab):-

        Depth > 0,
        NewDepth is Depth - 1,
        valid_moves(Tab, Player, MovesList),
        best(Tab, Player, State, MovesList, NewDepth, NextTab, NextVal), !
        ;
        length(Tab, Length),
        value(Tab, Player, NextVal, Length),
        NextTab = Tab.
```

Se estivermos numa profundidade maior que 0 (não estamos no último nível de expansão da árvore) então decrementamos a profundidade, recolhemos as jogadas válidas e de seguida percorremos essas mesmas jogadas, escolhendo a melhor. Se estivermos no último nível, apenas avaliamos o tabuleiro e retornamos o valor do mesmo.

O predicado que percorre as jogadas futuras e escolhe a melhor é o seguinte:

```
best(Tab, Player, State, [Move | NextMoves], Depth, BestTab, BestVal):-

        Move = [Row, Col, Dir],
        Piece is Player + 1,

        move([Row, Col, Dir], Piece, Tab, OutTab),

        nextTurn(Player, State, NextPlayer, NextState),

        minimax(OutTab, NextPlayer, NextState, Depth, Val1, _),

        best(Tab, Player, State, NextMoves, Depth, Tab2, Val2),

        betterOf(OutTab, Val1, Tab2, Val2, BestTab, BestVal, State).
```

Neste começamos por jogar a jogada que está na cabeça da lista das jogadas válidas e obtemos os valores referentes ao próximo turno.

De seguida chamamos novamente o predicado minimax, este vai expandir a árvore se não estiver no último nível e retornar o melhor valor dos futuros tabuleiros, senão apenas retorna o valor do tabuleiro atual.

De seguida chama o best, que vai avaliar as outras jogadas válidas, como isto é um processo recursivo este vai iterar todas as jogadas possíveis e quando chegar ao final delas (caso base de best) vai começar a comparar os valores das jogadas até chegar à inicial, que retornará o valor do melhor tabuleiro.

Infelizmente não conseguimos que o algoritmo funcionasse para profundidades maiores que dois, como não era algo pedido para o trabalho e como já tínhamos dedicado muito tempo a tentar resolver este problema decidimos não continuar a tentar, portanto esta secção ficou incompleta, deixámos o código (com comentários) deste algoritmo no trabalho, tal como a possibilidade de jogar contra bot que use este algoritmo (com profundidade máxima de dois).

3 Conclusões

O Neutreeko, mostrou-se um desafio grande que exigiu dedicar muito tempo. Conseguimos concluir tudo o que havia sido proposto, sendo que infelizmente não conseguímos concluir o algoritmo Minimax como desejávamos. Deparámo-nos com dificuldades em primeiro lugar no paradigma e no tempo que tivémos. Em relação às partes mais difíceis, salientamos o minimax, a robustez dos inputs e o display do jogo.

4 Referências

- Neutreeko
- Neutron
- Teeko
- Jan Kristian Haugland
- Minimax