Mbrane

Programação em Lógica 2019/2020 - Turma 2
3º Ano - 1º Semestre

Álvaro Francisco Barbosa Miranda - up201603694@fe.up.pt

Mestrado Integrado em Engenharia Informática e Computação

Índice

Introdução

Como objetivo deste trabalho foi-nos proposto a realização de um jogo de tabuleiro para dois jogadores em linguagem Prolog. Este envolve um tabuleiro, peças e várias regras de jogadas. Para além disso deverá permitir três modos de jogo sendo humano contra humano; humano contra computador e computador contra computador variando a dificuldade e quem começa a jogar.

Teremos então que realizar uma interface que permita o utilizador jogar de facto este jogo ou ver o computador jogar. Para isso irá usar-se modo de texto para representar isto.

O Jogo Mbrane

Mbrane é um jogo de tabuleiro moderno, de controlo de território para dois jogadores jogado num tabuleiro normal de sudoku. Foi criado em 2019 por Michael Nuell.

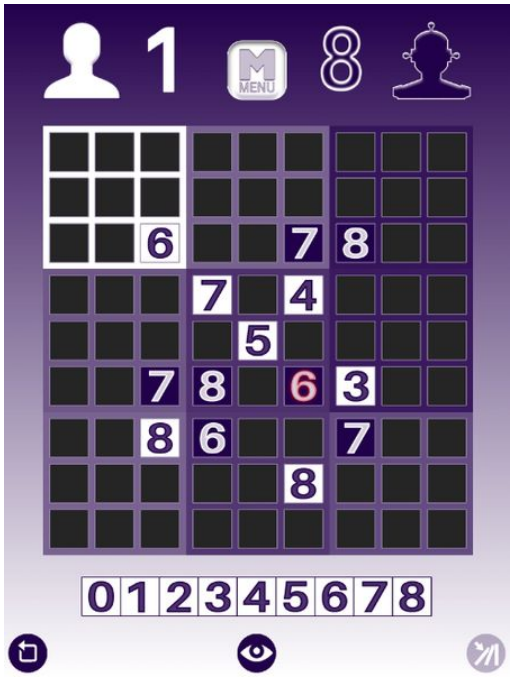


Figura 2.1 Print do jogo Mbrane da PlayStore

Regras

Tal como referido em cima é jogado num tabuleiro normal de Sudoku, isto é num quadrado 9x9, dividido em 9 quadrados mais pequenos 3x3.

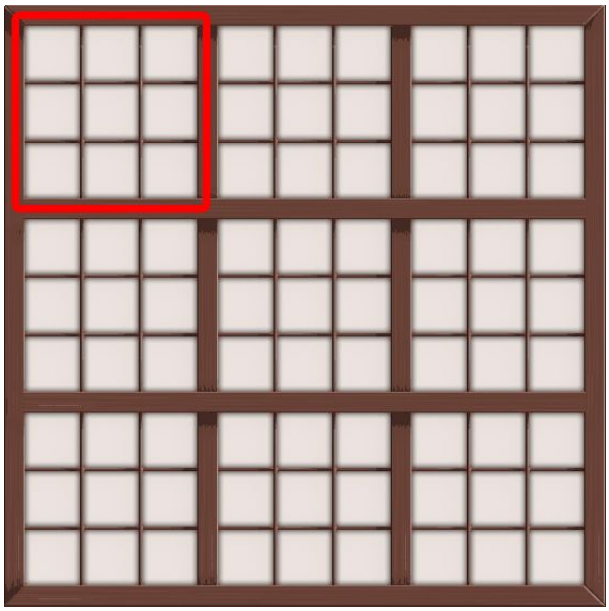


Figura 2.2 - Representação de um dos quadrados mais pequenos de um tabuleiro completamente vazio

No início do jogo o tabuleiro começa vazio, e em turnos, cada jogador pode meter um número entre 0 a 8 no tabuleiro. No entanto este colocamento de números segue as regras tradicionais do Sudoku. Estas são, na mesma linha e na mesma coluna não pode haver números repetidos. Também não pode existir números repetidos em nenhum dos 9 quadrados mais pequenos.

O jogo acaba quando por causa das regras normais do Sudoku não der para meter mais nenhum número ou o tabuleiro esteja completamente cheio.

Como referido em cima existem 9 quadrados 3x3 no tabuleiro. Cada um destes é uma região. Quando um jogador coloca um número numa casa ele ganha poder sobre essa região equivalente ao valor do número.

No entanto quando quando um número é colocado numa casa que toca ortogonalmente ou diagonalmente com uma casa de outra região, o jogador fica com influência sobre o poder dessa região. O valor dessa influência sobre essa região é metade do número colocado.

Figura 2.3 - O 7 na posição linha 3 e coluna 3 dá um poder de 7 à região onde está inserido. No entanto encontra-se na fronteira com mais três regiões e como tal vai fornecer a cada uma das três 3.5 pontos de influência ao jogador que a meteu

O objetivo do jogo é controlar mais regiões que o adversário. Controla-se as regiões tendo mais pontos que o adversário nesta região. Ou seja se o jogador 1 tem 13 pontos de Poder e 6 de influência o seu total de pontos nessa região vai ser 19. Se o jogador 2 tiver 11 de poder e 10 de influência fica com 21 pontos na região. Essa região fica então controlada pelo jogador 2 com a diferença dos pontos dele para o seu adversário, ou seja com 2 pontos.

No final quando não existem mais possibilidades de jogada, vai-se ver qual jogador controla mais regiões. No entanto aqui existem duas versões do jogo. Uma mais simples e outra mais complexa.

A versão mais simples apenas verifica consoante as regras apresentadas até agora os pontos de cada jogador em cada região e consequentemente o jogador que controlar mais regiões é o vencedor.

Na versão mais complexa quando as jogadas acabam vem a fase de Resolução. Nesta fase o que acontece vai ser que primeiro vai se procurar qual a região que tem maior poder absoluto. Ou seja independentemente do jogador qual das regiões tem maior poder. Nessa região vê-se então qual o jogador que a domina e todos os números do jogador adversário nessa região serão retirados. Isto irá influenciar a influência nas outras regiões. De seguida pega-se na região com o segundo maior valor absoluto e repete-se o procedimento. Os números do jogador que não a domina serão retirados. Isto é feio até à última região. No final desta fase o jogador que dominar mais regiões será o vencedor.

FIgura 2.4 - Tabuleiro Final de um jogo. No quadrado inferior direito como as peças do jogador preto já não contam pois esta está dominada pelo jogador branco e já vamos a meio da fase de Resolução.

Lógica do Jogo

Representação do Estado de Jogo e Visualização do Tabuleiro

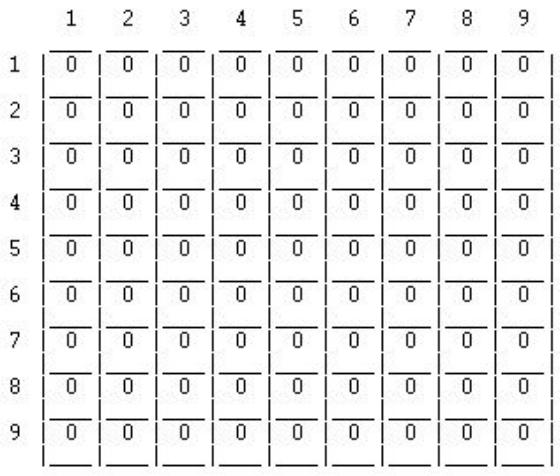
A representação interna do tabuleiro é feita numa lista de listas 9x9 de tamanho fixo. Começa tudo a 0 pois inicialmente o tabuleiro está vazio e o zero representa isso mesmo. Na versão original do jogo apenas se joga com os números de 0 a 8. No entanto após conferir com o professor e por não influenciar em quase nada, implementei com os números de 1 a 9 sendo que o número 0 representa uma casa vazia.

```
initialBoard([
  [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],
  [0,0,0,0,0,0,0,0,0],
  [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]
]).
```

Também para o jogador ter melhor perceção dos pontos que cada jogador tem em cada região criei um tabuleiro auxiliar, com as pontuações de cada região, guardado numa lista de listas 3x3.Tal como acima começa tudo a 0 pois está vazio.

```
initialPoints([
  [0.0,0.0,0.0],
  [0.0,0.0,0.0],
  [0.0,0.0,0.0]
]).
```

Figura 3.1- Representação interna dos dois tabuleiros



A visualização do tabuleiro ficará assim como na imagem ao lado sendo que o tabuleiro em cima é tabuleiro de jogo e o tabuleiro de baixo indica a pontuação de cada jogador em cada região. No iníco está tudo a zeros.

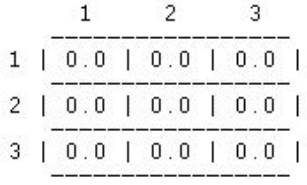


Figura 3.2 - Tabuleiros impresso no Sicstus de um tabuleiro inicial

As imagens seguintes demonstram um tabuleiro em que já não existem mais jogadas possíveis. Como podemos ver existem números negativos o que não faz sentido visto que apenas se jogam com os números de 1 a 9. No entanto isto faz parte da solução de implementação se decidiu. Os números positivos serão os que foram jogados pelo jogador número 1. Os negativos serão os jogados pelo jogador número 2. Para determinar se uma jogada será válida utiliza-se os valores absolutos destes números. Também para o tabuleiro da pontuação funcionará assim. Se for negativo é o jogador 2 que tem mais pontos nessa região. Se for o jogador 1 a pontuação será positiva.

```
initialBoard([
  [-6,7,-2,9,-1,0,3,-5,8],
  [-8,0,5,-7,2,6,-4,9,-1],
  [3,1,-9,4,5,-8,7,-6,0],
  [2,6,7,-5,0,9,-8,1,4],
  [0,5,-4,3,8,-2,6,0,-9],
  [0,9,-8,6,4,7,5,2,-3],
  [1,4,-6,8,7,5,9,-3,2],
  [7,-2,3,0,9,4,1,8,6],
  [9,-8,0,-2,6,-3,0,7,-5]
]).
```

	1	2	3	4	5	6	7	8	9
1	-6	7	-2	9	-1	0	3	-5	8
2	-8	0	5	-7	2	6	-4	9	-1
3	3	1	-9	4	5	-8	7	-6	0
4	2	6	7	-5	0	9	-8	1	4
5	0	5	-4	3	8	-2	6	0	-9
6	0	9	-8	6	4	7	5	2	-3
7	1	4	-6	8	7	5	9	-3	2
8	7	-2	3	0	9	4	1	8	6
9	9	-8	0	-2	6	-3	0	7	-5

```
initialPoints([
  [-11,1,12],
  [4,0,-4],
  [-8,-1,5]
]).
```

	1	2	3
1	-11	1	12
2	4	0	-4
3	-8	-1	5

Figura 3.3 - Respetiva representação de um estado final do jogo, tanto a sua representação interna como a representação na consola Sicstus

Um tabuleiro intermédio será qualquer tabuleiro após se ter feito a primeira jogada até não dar para fazer mais nenhuma jogada.

Para imprimir o tabuleiro na consola usamos as seguintes funções:

```
/*
* drawLine(+Line)
* prints a line of numbers
*/
drawLine([]).
drawLine([H | T]) :-
    H >= 0,
    H < 10,
    write(' '),
    write(H),
    write(' |'),
    drawLine(T).
drawLine([H | T]) :-
    H < -9,
    write(H),
    write('|'),
    drawLine(T).
drawLine([H | T]) :-
    write(' '),
    write(H),
    write('|'),
    drawLine(T).
```

Figura 3.4 - Função para imprimir cada número e semparadores

```
/*
* drawBoard(+Matrix, +Index)
* prints all numbers of Board
*/
drawBoard([], _).
drawBoard([H | T], Index) :-
    write('  '), write(Index), write(' |'),
    drawLine(H), nl,
    write('    |__|__|__|__|__|__|__|__|__|'), nl,
    NewIndex is Index+1,
    drawBoard(T, NewIndex).

/*
* drawTopSeparator
* prints the top border of Board
*/
drawTopSeparator:-
  write('      1   2   3   4   5   6   7   8   9  '), nl,
  write('     ___ ___ ___ ___ ___ ___ ___ ___ ___ '), nl.


/*
* drawCompleteBoard(+Board)
* prints the complete Board
*/
drawCompleteBoard(Board):-
  drawTopSeparator,
  drawBoard(Board, 1).
```

Figura 3.5 - Game Board Printing Functions

```
/*
* drawPoints(+Matrix, +Index)
* prints all numbers of Points Board
*/
drawPoints([], _).
drawPoints([H | T], Index) :-
    write('  '), write(Index), write(' |'),
    drawLine(H), nl,
    write('     ---------------- '), nl,
    NewIndex is Index+1,
    drawPoints(T, NewIndex).

/*
* drawTopSeparator
* prints the top border of Points board
*/
drawPointsSeparator:-
  write('       1     2     3  '), nl,
  write('     ---------------- '), nl.

/*
* drawPointsBoard(+Board)
* prints the complete Board of Points
*/
drawPointsBoard(Board1):-
  nl,nl,
  drawPointsSeparator,
  drawPoints(Board1, 1).
```

Figura 3.5 - Points Board Functions

Lista de Jogadas Válidas

De forma a obter uma lista de jogadas válidas é chamado o predicado `valid_moves(+Board, +Number, +IntermidiateMoves, -ListOfMoves).`

Este por sua vez vai chamar o predicado

`validMoveNumber(+Line, +Column, +Number, +Board, +MovesIntermediate, -MovesFinal).`

Para começar irei explicar a função validMoveNumber. Este recebe como argumentos um número a indicar a linha, um número a indicar a coluna, e um número a indicar qual o número que vai ser inserido na posição formada pelas duas coordenadas anteriores. Estas coordenadas

serão vistas no Board passado também como argumento. O argumento MovesIntermediate é uma lista auxiliar para guardar jogadas possíveis durante a execução da função. Então esta função é responsável por ver quais são as jogadas possíveis para um determinado número.

Para ver todas as jogadas possíveis indepedentemente do número é chamado a valid_moves que experimenta com os 9 números possíveis todas as jogadas possiveis para um determinado board, ou seja vai chamar a função validMoveNumber 9 vezes e retornará em ListOfMoves todas as jogadas possíveis na forma [Linha, Coluna, Número].

```
/*
* validMoveNumber(+Line, +Column, +Number, +Board, +MovesIntermediate, -MovesFinal)
* see all possible moves only to Number and puts all in Moves Final
*/
validMoveNumber(0, _, _, _, MovesIntermediate, MovesIntermediate).

validMoveNumber(Line, 0, Number, Board, MovesIntermediate, MovesFinal):-
  Line1 is Line - 1,
  validMoveNumber(Line1, 9, Number, Board, MovesIntermediate, MovesFinal).

validMoveNumber(Line, Column, Number, Board, MovesIntermediate, MovesFinal):-
  checkMovePossible(Line, Column, Number, Board, Boolean),
  Boolean =:= 1,
  append([Line], [Column], Coordinates),
  append(Coordinates, [Number], Moves),
  append(MovesIntermediate, [Moves], UpdatedList),
  Column1 is Column - 1,
  validMoveNumber(Line, Column1, Number, Board, UpdatedList, MovesFinal).

validMoveNumber(Line, Column, Number, Board, MovesIntermediate, MovesFinal):-
  Column1 is Column - 1,
  validMoveNumber(Line, Column1, Number, Board, MovesIntermediate, MovesFinal).
```

Figura 3.6 e 3.7 - predicados valid_moves e validMoveNumber

```
/*
* valid_moves(+Board, +Number, +IntermidiateMoves, -ListOfMoves)
* see all possible moves
*/
valid_moves(_, 0, CompleteMoves ,CompleteMoves).

valid_moves(Board, Number, IntermidiateMoves, ListOfMoves):-
  validMoveNumber(9, 9, Number, Board, [], MovesFinal),
  NewNumber is Number - 1,
  append(MovesFinal, IntermidiateMoves, UpdatedList),
  valid_moves(Board, NewNumber, UpdatedList, ListOfMoves).
```

Execução das Jogadas Válidas

Para a validação e execução de jogadas é usado os predicados. Neste caso não segui a recomendação de chamar move.

```
playerTurn(+Player, +Board, -NewBoard)

pcTurn(+Player, +Difficulty, +Board, +Points, +PreviousPoints, -NewBoard)
```

Para um jogador humano apenas é preciso saber qual o jogador que irá realizar a jogada e em qual Board irá jogar. O resto será pedido através de inputs para o jogador inserir. Falamos da linha e coluna onde quererá jogar e qual o número que irá jogar. Após tud inserido será feita a verificação se a jogada é válida ou não. Caso seja será posto no Board o número inserido na posição inserida. Caso não seja é pedido novamente para inserir valores.

```
* playerTurn(+Player, +Board, -NewBoard)
* request input from the player till it is a possible move and do the move
*/
playerTurn(Player, Board, Points, NewBoard):-
  repeat,
  clearSreen(30),
  drawCompleteBoard(Board),
  drawPointsBoard(Points),
  nl,write('Player '), write(Player),  write(' Turn'),
  nl,write('Insert Line:'),
  readImput(Line), nl,
  write('Insert Column:'),
  readImput(Column),nl,
  write('Insert Number:'),
  readImput(Number),nl,!,
  checkMovePossible(Line,Column, Number, Board, Boolean),
  Boolean =:= 1,
  putNumber(Line, Column, Player, Number, Board, NewBoard).
```

Figura 3.8 - Predicado playerTurn

Para o computador como não será necessário o input basta ver quais são as jogadas válidas e escolher uma delas. Isto é feito no predicado choose_moves que iremos ver melhor mais à frente.

A função split serve para tirar o move do formato [Line, Column, Number] para de seguida realizar a jogado no Board dado como argumento.

```
/*
* pcTurn(+Player, +Difficulty, +Board, +Points, +PreviousPoints, -NewBoard)
* see possible moves and choose a random in easy mode or the best move if in hard mode
*/
pcTurn(Player, Dificulty, Board, Points, PreviousPoints, NewBoard):-
  clearSreen(30),
  drawCompleteBoard(Board),
  drawPointsBoard(PreviousPoints),
  nl,write('PC '), write(Player),  write(' Turn'), nl,
  write('Press Key To Let Pc Play'),
  pressLetter,

  choose_move(Board, Points, Dificulty, Player, Move),
  split(Move, Line, Column, Number),
  putNumber(Line, Column, Player, Number, Board, NewBoard).
```

Figura 3.9 - Predicado pcTurn

É importante ver também melhor o predicado que verifica de facto se uma jogada é válida ou não. É ele :

`checkMovePossible(+Line, +Column, +Number, +Board, -Boolean)`

Primeiro verifica se tanto a linha, coluna e número são números válidos. Após isso verifica se a posição está vazia. De seguida verifica as regras básicas do sudoku. Primeiro vê se não nenhum número igual na mesma coluna, depois na mesma linha e no final no mesmo quadrado. Se passar nestes testes todos é uma jogada válida.

```
/**
* checkMovePossible(+Line, +Column, +Number, +Board, -Boolean)
* See if the move of puting the Number in Line and Column of Boart is possible, Boolean is 1 if yes and 0 if no
*/
checkMovePossible(Line, Column, Number, Board, Boolean):-
  Line > 0,
  Line < 10,
  Column > 0,
  Column < 10,
  Number > 0,
  Number < 10,
  getMatrixAt(Line,Column, Board, Numb),
  Numb =:= 0,
  checkColumns(Column, Number, Board),
  checkLine(Line, Number, Board),
  checkLitleSquares(Line, Column, Number, Board),
  Boolean is 1.
checkMovePossible(_, _, _, _, Boolean):-
  Boolean is 0.
```

Figura 3.9 - Predicado checkMovePossible

Final do Jogo

Na verificação final do jogo é usado o predicado:

`game_over(+Points, -Winner)`

Este recebe o board de Pontos que usando o predicado regionsOnwed irá ver quantas regiões cada jogador controla no final do jogo(F1 as que o jogador 1 controla, F2 as que o jogador 2 controla), quando não há mais jogadas possíveis. O jogador que controlar mais zonas será o vencedor. A função flatten2 é só uma função auxiliar que transforma uma lista de lista em apenas uma lista seguida.

```
/*
* game_over(+Points, -Winner)
* Winner is the player that has more regions controled
*/
game_over(Points, Winner):-
  flatten2(Points, FinalList),

  regionsOnwed(FinalList, 0, 0, F1, F2),

  write('Score'), nl,
  write('Player1: '), write(F1), nl,
  write('Player2: '), write(F2), nl,
  (F1 > F2, Winner is 1);
  (F2 > F1, Winner is 2);
  Winner is 0.
```

Figura 3.10 - Predicado game_over

Avaliação do Tabuleiro

A heurística usada para avaliação do tabuleiro é bastante elementar. Apenas vê para cada jogador qual a soma de pontos que ele tem no momento. O seu objetivo é maximizar este valor para que no final controle mais zonas que o seu adversário.

```
/*
* value(+Points, +Player, -Value)
* the value is the amount of points a player has
*/
value(Points, 1, Value):-
  matrixSum(Points, Sum),
  Value is Sum.

value(Points, 2, Value):-
  matrixSum(Points, Sum),
  Value is 0 - Sum.
```

Figura 3.11 - Predicado value

Jogada do Computador

Para escolha de jogada do computador existe o predicado

`choose_move(+Board, +Points, +Dificulty, +Player, -Move)`

Que por sua vez caso a dificuldade seja 2 que indica a dificuldade dificil irá chamar o predicado

`chooseBestMoves(+Board, +Points, +Player, +AllMoves, +IntermidiateMoves, +IntermidiateValue, -BestMoves)`

Caso seja a dificuldade 1(fácil) apenas vai ver quais são as jogadas válidas e escolher uma aleatória dentro dessas.

Caso a dificuldade seja 2(dificil) irá em vez de só ver quais as jogadas válidas ver quais são as jogadas que beneficiarão mais o jogador atual. Para isso usa a função já referida que vê para cada jogada válida qual o value que o tabuleiro ficará caso essa jogada seja feita. Para isso para cada jogada que se preveja é guardado o maior value que é possível alcançar. Ao iterar-se pelas jogadas se com uma se alcançar um value superior ao maior alcançado anteriormente guarda-se essa jogada e são descartadas as anteriormente guardadas. Caso o value seja iguar ao maior previamente alcançado junta-se à lista de melhores moveimentos. Caso seja inferior ao maior value a jogada é descartada. Assim esta função devolve uma lista com os movimentos que maximizam mais o value para o jogador na próxima jogada.

```
/*
* choose_move(+Board, +Points, +Dificulty, +Player, -Move)
* choose a Move acording to the board and the dificulty
*/
choose_move(Board, _, 1, _, Move):-
  valid_moves(Board , 9 , [], ListOfMoves),
  length(ListOfMoves, Length),
  random(0, Length, Random),
  nth0(Random, ListOfMoves, Move).

choose_move(Board, Points, 2, Player, Move):-
  valid_moves(Board , 9 , [], ListOfMoves),
  chooseBestMoves(Board, Points, Player, ListOfMoves, [], 0, BestMoves),
  length(BestMoves, Length),
  random(0, Length, Random),
  nth0(Random, BestMoves, Move).
```

Figura 3.12 Predicado choose_move

```
/*
* chooseBestMoves(+Board, +Points, +Player, +AllMoves, +IntermidiateMoves, +IntermidiateValue, -BestMoves)
* choose the best move acording to the Board
*/
chooseBestMoves(_, _, _, [], BestMoves, _, BestMoves).

chooseBestMoves(Board, Points, Player, [Move|T], IntermidiateMoves, IntermidiateValue, BestMoves):-
  split(Move, Line, Column, Number),
  putNumber(Line, Column, Player, Number, Board, NewBoard),
  updateAllPoints(NewBoard, Points, Points2),
  value(Points2, Player, Value),!,
  (
    (Value > IntermidiateValue,
    append([Move], [], NewBestMoves),
    chooseBestMoves(Board, Points, Player, T, NewBestMoves, Value, BestMoves));

    (Value = IntermidiateValue,
    append(IntermidiateMoves, [Move], NewBestMoves),
    chooseBestMoves(Board, Points, Player, T, NewBestMoves, Value, BestMoves));

    chooseBestMoves(Board, Points, Player, T, IntermidiateMoves, IntermidiateValue, BestMoves)
  ).
```

Figura 3.13 - Predicado chooseBestMoves

Conclusões

O trabalho deu muito trabalho visto que tive que o realizar sozinho. No entanto apenas um aspecto muito pequeno do jogo não foi realizado. Mas todas as funções requisitadas estão implementadas e funcionam sem erros.

Considero que consolidei o que temos vindo a aprender nas aulas no que se destaca a programação lógica. Sendo o principal destaque o backtrack que muitas vezes nos exige que pensemos ao contrário do que normalmente estamos habituados.

No entanto há medida que vamos avançando na construção de predicados e consequentemente do jogo torna-se mais fácil realizar mais predicados.

Bibliogafia

https://www.boardgamegeek.com/boardgame/278660/mbrane