Inteligência Artificial

Problema 3: Tic-Tac-Toe

Docente: José Valente de Oliveira

Discentes: Afonso Silva nº 76943, Daniel Andrade nº76922

Problema

Pretende-se criar uma IA que, dado um tabuleiro m colunas por n linhas, consiga ganhar ao utilizador num jogo de tic-tac-toe onde a win condition é ter k simbolos em linha, quer horizontal, vertical ou diagonalmente alinhados. Isto com m, n e k sendo números inteiros.

MinMax

O algoritmo Minimax é bastante útil em jogos de decisão, como xadrez ou tic-tac-toe, para encontrar a melhor jogada possível de um jogador, considerando que o adversário também está a tentar maximizar o seu resultado. O nome "Minimax" refere-se à minimização das perdas do jogador e à maximização dos ganhos, tendo em conta todas as possíveis jogadas.

O funcionamento do algoritmo é baseado em árvores de decisão. Cada nó na árvore representa um estado do jogo, e as arestas ligam esses estados às possíveis jogadas seguintes. Os nós folha representam os estados terminais do jogo, nos quais é possível determinar o resultado (vitória, derrota ou empate).

O algoritmo percorre recursivamente a árvore de decisão, alternando entre os jogadores (um jogador tenta maximizar o resultado, enquanto o outro tenta minimizá-lo). Em cada nível, o algoritmo avalia as possíveis jogadas, atribuindo um valor a cada nó. O valor é calculado retroativamente para os nós superiores da árvore,isto é, o valor de cada jogada é determinado olhando para o resultado final do jogo (nós folha) e, em seguida, retrocede-se pela árvore, escolhendo os valores máximos para os nós em que é a vez do jogador maximizar e os valores mínimos para os nós em que é a vez do jogador minimizar.

Se um jogador é maximizador, ele escolhe a jogada que leva ao nó filho com o maior valor. Se

é minimizador, escolhe a jogada que leva ao nó filho com o menor valor. Este processo continua até chegar ao nó raiz, representando a melhor jogada possível para o jogador maximizador, considerando as ações do jogador minimizador.

O algoritmo Minimax é eficaz para encontrar a melhor jogada em jogos de decisão com informação completa, onde todas as informações relevantes estão disponíveis. No entanto, pode tornar-se impraticável em jogos mais complexos devido à explosão combinatória do número de estados possíveis. Algumas otimizações, como a poda alfa-beta, são frequentemente utilizadas para tornar o algoritmo mais eficiente.

Esta poda alfa-beta é usada para reduzir a quantidade de nós avaliados numa árvore de decisão, mantendo os mesmos resultados. Acaba por ser bastante útil quando a árvore se torna demasiado complexa. A ideia fundamental da poda alfa-beta é eliminar a avaliação de certos ramos da árvore que sabemos que não levarão a uma solução melhor do que a já encontrada. Os parâmetros "alfa" e "beta" representam intervalos que indicam os valores mínimos e máximos que um jogador maximizador ou minimizador, respetivamente, já encontrou ao longo do caminho.

Ao avaliar os nós da árvore, se um jogador encontra um ramo que leva a uma solução inferior ao intervalo existente (alfa-beta), esse ramo pode ser podado, pois não influenciará a escolha final. Esta eliminação de ramos desnecessários reduz drasticamente o número de nodos a serem avaliados, tornando o algoritmo mais eficiente.

Heuristica

A heurística como visto no trabalho anterior é uma forma de “prever” o melhor caminho. No mixmax esta tem uma funcionalidade um pouco diferente, aqui nos estamos a jogar contra um jogador e o comportamento da árvore vai sempre diferir dependendo das jogadas de cada jogador. Assim é impossível existir uma heurística perfeita. O nosso objetivo com a heurística é calcular o potencial da jogada, ou seja, o quanto ela beneficia o jogador. Enquanto a avaliação de uma jogada é a subtração entre o potencial do jogador X menos o potencial do jogador O.

O potencial de um jogador pode ser calculado da seguinte forma, verificamos se existem pelo menos k espaços vazios na diagonal direita, diagonal esquerda, vertical e horizontal, num determinado índice. Se não existir potencial de ganhar então o resultado é 0, se houver o potencial é o valor de turnos consecutivos. Por exemplo:

	X	

Assumindo que k = 3, no exemplo acima podemos verificar que o potencial é 4, isto pois X pode ganhar na vertical, horizontal, diagonal esquerda e diagonal direita. O valor final é só quatro pois não temos mais nenhum X, se por exemplo tivéssemos outro X na diagonal o potencial seria 5 pois teríamos mais um X.

		O
	X	

Se por exemplo o adversário jogar e tentar impedir-nos, o nosso potencial desce de 4 para 3 pois agora

só é possível ganhar na vertical, horizontal e diagonal direita.

Agora vemos a heurística em pseudo código.

```
getPotentialWin(row, column, turn, jumpRow, jumpColumn)
        result = 0
        count = 0

        i = row
        j = column

        minRow = min(this.rows - 1, row + ((this.winningCondition - 1) * |jumpRow|))
        minColumn = min(this.columns - 1, column + ((this.winningCondition - 1) * |jumpColumn|)

        maxRow = max(0, row - ((this.winningCondition - 1) * |jumpRow|))
        maxColumn = max(0, column - ((this.winningCondition - 1) * |jumpColumn|))
…
```

O método *getPotentialWin* vai receber como argumento, a linha e coluna onde se localizada o índice que queremos calcular o potencial, e o "salto" que temos que dar nas linhas e colunas e o turno para o qual queremos calcular o potencial. Os saltos representam se queremos verificar na diagonal, vertical ou horizontal. Por exemplo se queremos verificar na horizontal então a linha vai ser sempre a mesma, mas temos que andar de coluna em coluna para analisar-mos o potencial de uma win. Inicializamos 6 variáveis, result que como o nome indica é o resultado do nosso método, count que ajuda a sabermos se existe uma win no índice dado, i e j guardam a linha e a coluna respetivamente, minRow e minColumn são o limite para o qual podemos "andar para a frente" nas linhas e colunas respetivamente do array, por fim maxRow e maxColumn são iguais ás anteriores mas estas "andam para trás".

```
 …
While i >= 0 && i <= minRow & j >= 0 && j <= minColumn
        If getOpositePlayer(turn) == board[i][j]
           break
        else if turn == board[i][j]
           result++
        count++

        i += jumpRow
        j += jumpColumn
…
```

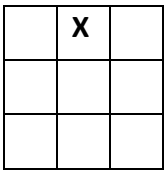
Assumindo que o player a jogar é X. A seguir criamos um while onde o i tem que ser maior ou igual a zero e menor ou igual a minRow e para J igual. Vamos percorrer o array "para a frente" até atingirmos o falor de minRow ou minColumn. Por cada X que encontrarmos fazemos result++, se encontrarmos algum O saímos do loop, e fazemos sempre count++ e adicionamos os respetivos jumps as variáveis corretas. Assim sabemos quanto quantos valores disponíveis ou iguais a X temos "para a frente", mas agora precisamos verificar "para trás".

Reiniciamos as variáveis i e j. Criamos o while onde i tem que ser que ser maior ou igual a maxRow e menos que todas as linhas, para j é igual, mas desta vez fazemos a mesma coisa só que para trás. No final retornamos o result se o count foi maior ou igual a k.

```
 …
return count >= k ? result : 0
```

```
…
 i = row - jumpRow
 j = column - jumpColumn
 while i >= maxRow && i < this.rows && j >= maxColumn && j < this.columns
    if getOpositePlayer(turn) == board[i][j]
       break
    else if turn == board[i][j]
       result++
    count++

    i -= jumpRow
    j -= jumpColumn
…
```



Se jogarmos na posição acima e verificarmos qual o seu potencial, chegamos á conclusão que é 3. Vamos pegar no exemplo da horizontal e aplica-lo com o método anterior. Começamos com a row = 0 e column = 1, o turno é X e os saltos são, jumpRow = 0 e jumpColumn = 1, isto pois nós queremos mover-nos apenas na linha para trás e para a frente. Seguinda a ordem da função começamos por inicializar as variáveis que vão te os seguintes valores: i = 0, j = 1, minRow = 0, minColumn = min(2, 3), maxRow = 0, maxColumn = max(0, -1). Com estes valores podemos avançar, entramos no ciclo while onde encontramos um "if" e "else if", onde vamos ignorar o primeiro pois a posição atual é [0, 1] e o turno que está na posição é X, ou seja, entramos no else if, onde fazemos a operação de result++, e logo em seguida count++, i += 0, j += 1. Podemos reparar que o i nunca sai do sítio pois não queremos mover a linha. Andamos então uma posição para a direita, agora estamos na posição [0, 2], não entramos em nenhum "if" então apenas adicionamos 1 ao count, e fazemos a adição dos jumps. O ciclo então acaba, pois a posição atual seria [0, 3], mas j tem q ser menos ou igual a 2. Avançamos no código pois ainda não acabou, agora temos de ir para trás, para trás é a mesma coisa, mas desta vez subtraímos as posições. No final temos count = 3, e result = 1 , então como count é igual a k retornamos o result e é 1, se fizermos o resto para a vertical também, obtemos no total 2 de potencial. Concluimos que a heurista então é a soma de todos os potenciais de todos os turnos que estamos a calcular a heurística.

Avaliação

A avaliação é simples, retornamos a heurística do jogador atual menos a heurística do jogador oposto, mas tem um porem, se nessa jogada o jogador perdeu então queremos retornar um valor negativo alto, se o jogador ganhou queremos retornar um valor positivo alto. Isto deve-se ao facto da natureza do MinMax, ao retornarmos uma avaliação negativa alta, esse nó nunca vai ser escolhido na árvore, ou seja, é quase impossível perder, pois o nosso "IA" não escolhe jogadas que façam ele perder, a forma de dar a volta a isso seria tentar fazer jogadas onde encurralamos o oponente, não importa onde o adversário jogue ele vai perder, mas este estilo de jogo pode ter um simples contra-ataque, aumentamos a depth, do minmax, assim vamos saber quase todas as jogadas possíveis e escolhemos a que nos beneficia mais. Se retornarmos um valor muito alto positivo o "AI " joga onde ganha, assim garantimos a vitória.

Em código o método da avaliação teria o seguinte formato,

```
getEvaluation(ID turn)
    if this.winner == turn
        return 999999
    else if this.winner == getOpositePlayer(turn)
        return -999999

    return getHeuristic(turn) - getHeuristic(getOpositePlayer(turn))
```

Otimizações

Vamos agora testar a eficiência do nosso "IA", se não tentarmos otimizar de alguma forma obtemos os seguintes tempos,

Depth	Tempo de execução(ms)
2	9.6
4	44.9
6	500.13333
8	7884.6333
10	42632.0
12	117690.0

***Tabela 1**. Jogo 4x4

Nota: Os tempos acima foram feitos com o "IA" a jogar contra ele mesmo. Os jogos acabaram todos em empate. Isto dá-se para todos os jogos e tabelas que conterem um asterisco ''.

Como podemos ver na tabela acima nas depths baixas o "IA" não é lento, até podemos dizer que é rápido, mas isto o tempo começa a ser um problema com depths altas, isto é mau, pois nós queremos depths mais altas possíveis. Existem algumas formas de resolver o problema da otimização, a primeira que podemos implementar é a cache.

A cache é implementada diretamente no minmax. Criamos então um hashMap onde a chave é o layout e o valor é a avaliação. A hash implementada no layout não é a melhor mas também não é a pior, esta funciona da seguinte forma.

	X	

Se usarmos o exemplo anterior, pode pensar da seguinte forma, um espaço vazio equivale ao valor 0, o X ao valor 1 e o valor O equivale a 2, com isto podemos formar um número de base 3, ou seja, usando exemplo acima o valor calculado em base 3 seria 000 000 010 convertendo para decimal seria 1 * 3^(1) , ou seja, 3. Se por exemplo adicionarmos um circulo na primeira posição, ou seja,

O	**X**	

Agora a hash seria a soma da hash anterior, com a hash atual da ultima posição jogada, ou seja, 000 000 010 + 000 000 001 = 000 000 011, convertendo para decimal a hash é igual a 4.

Sabendo como a hash funciona podemos criar a cache. A cache vai guardar o layout com a sua correspondente avaliação, neste casa não é a sua avaliação "direta", mas sim o valor que chegou das folhas da árvore e veio até esse layout. Quando encontramos algum layout que já existe dizemos que a avaliação daquele nó é avaliação do que está na cache. Em pseudo código temos o seguinte,

```
…
        evaluation = 0;
        if !cache.containsKey(child)
            evaluation = minmax(…)
            cache.put(child, evaluation)
         else
            evaluation = cache.get(child)
...
```

Será que os tempos ficaram melhores? Vejamos.

Depth	Tempo de execução(ms)
2	8.533334
4	35.9
6	246.5
8	2240.4666
10	4370.1333
12	3948.5334
14	3249.4333
16	2361.1667

***Tabela 2**. Jogo 4x4

Se compararmos as duas tabelas vemos um aumento enorme em eficiência, em depth de 10 conseguimos diminuir o tempo 10x o seu original! Podemos ver que os dados para as depths de 12 para baixo não existem, isto deve-se ao facto de serem tão lentos que não vale a pena obtermos os dados para comparar.

Um ponto interessante é que podemos verificar que a partir da depth 10 para baixo na **tabela 2**, os valores vão diminuindo, isto deve-se a um facto curioso que vai ser explicado mais á frente.

Mas ainda podemos melhorar o código coma ajuda das simetrias e rotações. Tabuleiros simétricos são tabuleiros que são iguais eles apenas são apresentados de forma diferente assim como tabuleiros "rodados", vejamos este exemplo,

X		

		X

No exemplo acima vemos uma rotação para a direita. Se por exemplo tentarmos fazer a avaliação dos dois, elas são as mesmas, então não queremos gerar tabuleiros iguais para pouparmos tempo, por exemplo se um tabuleiro começa vazio os dois exemplos acima seria duas das dezasseis "children" do tabuleiro então não geramos o segundo, assim geramos menos 1 ramo da árvore, talvez consigamos poupar imenso tempo. Isto também se aplica com as simetrias, se juntarmos os dois conseguimos descobrir todos os tabuleiros iguais possíveis. Se quisermos saber se um tabuleiro é igual a outro simetricamente apenas precisamos de rodalo três

vezes, em seguido fazemos a simetria do tabuleiro original e rodamos outra vez 3 vezes, podemos usar qualquer simetria, horizontal, vertical, diagonal esquerda ou direita. Mas são sempre 8 variações do mesmo tabuleiro. Assim conseguimos criar algum método que verifique se o tabuleiro é igual a outro simetricamente. Com isto em mente basta apenas aplicar esse método criado no método children assim só gerando os filhos únicos. Vejamos então os resultados,

Depth	**Tempo de execução(ms)**
2	6.9333334
4	34.2
6	232.56667
8	1690.9667
10	3800.8
12	3236.0334
14	2465.4333
16	1712.5

***Tabela 3**. Jogo 4x4

Os resultados são um pouco despontadores…, temos uma pequena melhoria de tempo, que é boa, mas não como o anterior, se compararmos a melhoria antiga com a nova na depth 10, na anterior temos um aumento de 10x mais eficiência enquanto que com simetrias apenas temos uma melhoria de 15%, mas todas as otimizações são bem-vindas.

Comportamento

Como vimos no tópico anterior o tempo começa a ficar mais rápido a partir de depth 10, isto deve-se ao facto que o "IA" sabe que o jogo vai empatar. A avaliação de muitos nós vais ser zero pois chegamos ao fim da árvore onde vemos que o resultado mais certo será empate logo, haverá imensos cortes de alfa beta devida a este fenómeno tornando assim o "IA" mais eficiente, vejamos o resultado de um jogo com depth 16,

X	**O**	**X**	**O**
X	**O**	**X**	**O**
X	**X**	**O**	**X**
O	**O**	**X**	**O**

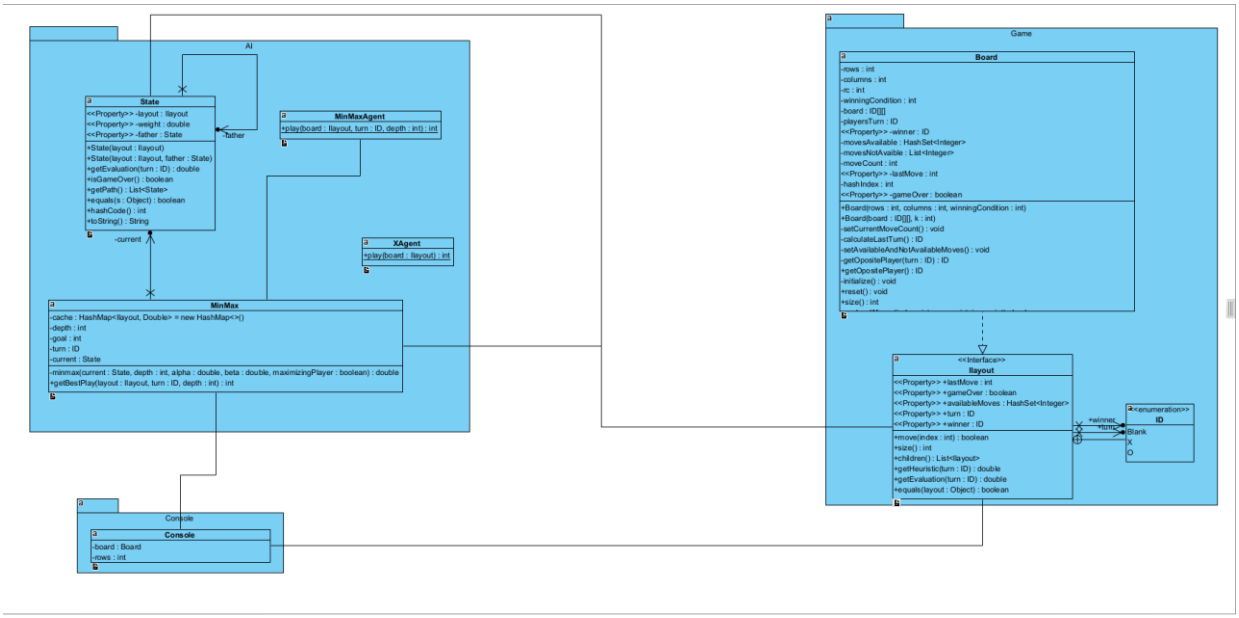
O "IA" simplesmente vai jogar na posição mais perto, mas se assim continuasse da mesma forma alguém ganhava então o "IA" mudo de comportamento para impedir alguma win.

...

Conclusão

Podemos então concluir que o algoritmo minmax é uma excelente escolha para crirar um “IA” de um tabuleiro onde existe um jogador contra outro, é fácil de perceber e implementar e diversas otimizações podem ser feitas.
A otimização que salva o nosso tempo é a cache, as simetrias são interessantes mas quase insignificantes para a eficiência do “IA”, percebemos também um comportamento muito interessante no “IA” dependendo da depth.

UML



Tests

```
@Test
public void DrawTest(){

    Ilayout.ID array[][] = {
        {Ilayout.ID.X,     Ilayout.ID.X,     Ilayout.ID.X,     Ilayout.ID.O},
        {Ilayout.ID.O,     Ilayout.ID.X,     Ilayout.ID.X,     Ilayout.ID.X},
        {Ilayout.ID.O,     Ilayout.ID.O,     Ilayout.ID.X,     Ilayout.ID.O},
        {Ilayout.ID.O,     Ilayout.ID.X,     Ilayout.ID.O,     Ilayout.ID.Blank},
        };

    Board board = new Board(array, 4);
    board.move(15);

    assertTrue(board.isGameOver());
    assertEquals(board.getWinner().name(), Ilayout.ID.Blank.name());
}
```

```
    @Test
    public void WinnerHorinzotalLineTest(){

        Ilayout.ID array[][] = {
            {Ilayout.ID.X,     Ilayout.ID.X,     Ilayout.ID.X,     Ilayout.ID.Blank},
            {Ilayout.ID.Blank,  Ilayout.ID.Blank, Ilayout.ID.Blank, Ilayout.ID.Blank},
            {Ilayout.ID.O,     Ilayout.ID.O,     Ilayout.ID.Blank, Ilayout.ID.Blank},
            {Ilayout.ID.O,     Ilayout.ID.Blank, Ilayout.ID.Blank, Ilayout.ID.Blank},
            };

        Board board = new Board(array, 4);
        board.move(3);

        assertTrue(board.isGameOver());
        assertEquals(board.getWinner().name(), Ilayout.ID.X.name());
    }

    @Test
    public void WinnerVerticalLineTest(){

        Ilayout.ID array[][] = {
            {Ilayout.ID.X,       Ilayout.ID.O,       Ilayout.ID.Blank,    Ilayout.ID.Blank},
            {Ilayout.ID.X,       Ilayout.ID.O,       Ilayout.ID.O,        Ilayout.ID.Blank},
            {Ilayout.ID.X,       Ilayout.ID.Blank,    Ilayout.ID.Blank,    Ilayout.ID.Blank},
            {Ilayout.ID.Blank,    Ilayout.ID.Blank,    Ilayout.ID.Blank,    Ilayout.ID.Blank},
            };

        Board board = new Board(array, 4);
        board.move(12);

        assertTrue(board.isGameOver());
        assertEquals(board.getWinner().name(), Ilayout.ID.X.name());
    }

    @Test
    public void WinnerDiagonalRightILineTest(){

        Ilayout.ID array[][] = {
            {Ilayout.ID.X,       Ilayout.ID.O,       Ilayout.ID.Blank,     Ilayout.ID.O},
            {Ilayout.ID.Blank,    Ilayout.ID.X,       Ilayout.ID.Blank,     Ilayout.ID.Blank},
            {Ilayout.ID.Blank,    Ilayout.ID.Blank,    Ilayout.ID.X,        Ilayout.ID.Blank},
            {Ilayout.ID.O,       Ilayout.ID.Blank,    Ilayout.ID.Blank,     Ilayout.ID.Blank},
            };

        Board board = new Board(array, 4);
        board.move(15);

        assertTrue(board.isGameOver());
        assertEquals(board.getWinner().name(), Ilayout.ID.X.name());
    }
}
```

```
@Test
  public void WinnerDiagonalLeftlLineTest(){

    Ilayout.ID array[][] = {
      {Ilayout.ID.Blank,    Ilayout.ID.Blank,    Ilayout.ID.Blank,    Ilayout.ID.X},
      {Ilayout.ID.Blank,    Ilayout.ID.Blank,    Ilayout.ID.X,        Ilayout.ID.O},
      {Ilayout.ID.Blank,    Ilayout.ID.X,        Ilayout.ID.O,        Ilayout.ID.Blank},
      {Ilayout.ID.Blank,    Ilayout.ID.O,        Ilayout.ID.Blank,    Ilayout.ID.Blank},
      };

    Board board = new Board(array, 4);
    board.move(12);

    assertTrue(board.isGameOver());
    assertEquals(board.getWinner().name(), Ilayout.ID.X.name());

  }

  @Test
  public void FailDiagonalLeftlLineTest(){

    Ilayout.ID array[][] = {
      {Ilayout.ID.X,       Ilayout.ID.O,        Ilayout.ID.Blank,    Ilayout.ID.O},
      {Ilayout.ID.Blank,    Ilayout.ID.Blank,    Ilayout.ID.X,        Ilayout.ID.Blank},
      {Ilayout.ID.Blank,    Ilayout.ID.X,        Ilayout.ID.Blank,    Ilayout.ID.Blank},
      {Ilayout.ID.Blank,    Ilayout.ID.Blank,    Ilayout.ID.O,        Ilayout.ID.Blank},
      };

    Board board = new Board(array, 4);
    board.move(12);

    assertFalse(board.isGameOver());

  }

  @Test
  public void WinnerHorinzotalLineTestDiffSize(){
    Ilayout.ID array[][] = {
      {Ilayout.ID.X,       Ilayout.ID.X,        Ilayout.ID.Blank,    Ilayout.ID.Blank},
      {Ilayout.ID.O,       Ilayout.ID.Blank,    Ilayout.ID.Blank,    Ilayout.ID.Blank},
      {Ilayout.ID.O,       Ilayout.ID.Blank,    Ilayout.ID.Blank,    Ilayout.ID.Blank},
      {Ilayout.ID.Blank,    Ilayout.ID.Blank,    Ilayout.ID.Blank,    Ilayout.ID.Blank},
      };

    Board board = new Board(array, 3);
    board.move(2);

    assertTrue(board.isGameOver());
    assertEquals(board.getWinner().name(), Ilayout.ID.X.name());
  }
```

```
Test
   public void WinnerVerticalLineTestDiffSize(){
      Ilayout.ID array[][] = {
         {Ilayout.ID.X,       Ilayout.ID.O,       Ilayout.ID.Blank,     Ilayout.ID.Blank},
         {Ilayout.ID.X,       Ilayout.ID.O,       Ilayout.ID.Blank,     Ilayout.ID.Blank},
         {Ilayout.ID.Blank,    Ilayout.ID.Blank,    Ilayout.ID.Blank,     Ilayout.ID.Blank},
         {Ilayout.ID.Blank,    Ilayout.ID.Blank,    Ilayout.ID.Blank,     Ilayout.ID.Blank},
         };

      Board board = new Board(array, 3);
      board.move(8);

      assertTrue(board.isGameOver());
      assertEquals(board.getWinner().name(), Ilayout.ID.X.name());
   }

   @Test
   public void WinnerDiagonalRightLineTestDiffSize(){
      Ilayout.ID array[][] = {
         {Ilayout.ID.X,       Ilayout.ID.O,       Ilayout.ID.Blank,     Ilayout.ID.Blank},
         {Ilayout.ID.Blank,    Ilayout.ID.X,       Ilayout.ID.Blank,     Ilayout.ID.Blank},
         {Ilayout.ID.O,       Ilayout.ID.Blank,    Ilayout.ID.Blank,     Ilayout.ID.Blank},
         {Ilayout.ID.Blank,    Ilayout.ID.Blank,    Ilayout.ID.Blank,     Ilayout.ID.Blank},
         };

      Board board = new Board(array, 3);
      board.move(10);

      assertTrue(board.isGameOver());
      assertEquals(board.getWinner().name(), Ilayout.ID.X.name());
   }

   @Test
   public void WinnerDiagonalLeftLineTestDiffSize(){
      Ilayout.ID array[][] = {
         {Ilayout.ID.O,       Ilayout.ID.Blank,    Ilayout.ID.Blank,     Ilayout.ID.X},
         {Ilayout.ID.Blank,    Ilayout.ID.Blank,    Ilayout.ID.X,       Ilayout.ID.Blank},
         {Ilayout.ID.Blank,    Ilayout.ID.Blank,    Ilayout.ID.Blank,     Ilayout.ID.O},
         {Ilayout.ID.Blank,    Ilayout.ID.Blank,    Ilayout.ID.Blank,     Ilayout.ID.Blank},
         };

      Board board = new Board(array, 3);
      board.move(9);

      assertTrue(board.isGameOver());
      assertEquals(board.getWinner().name(), Ilayout.ID.X.name());
   }
```

...

Bibliografia

https://www.youtube.com/watch?v=l-hh51ncgDI&t=107s