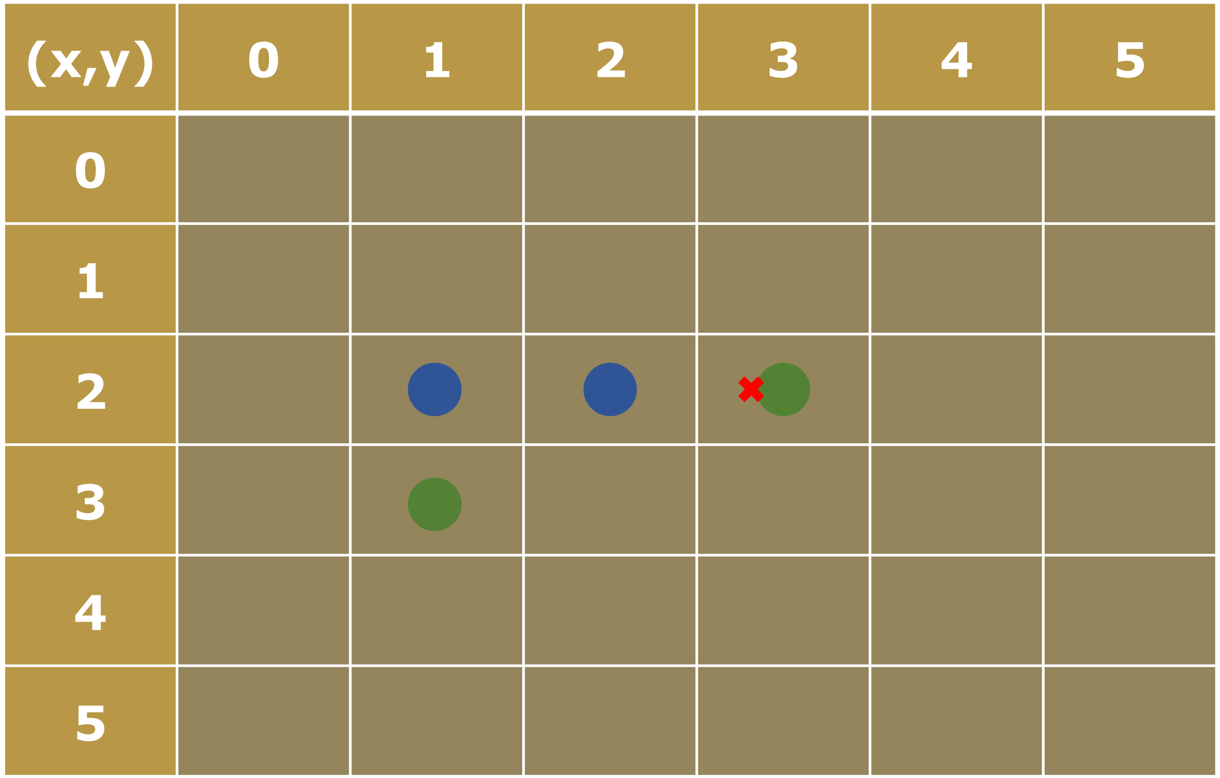
Gekitai(Push Away)

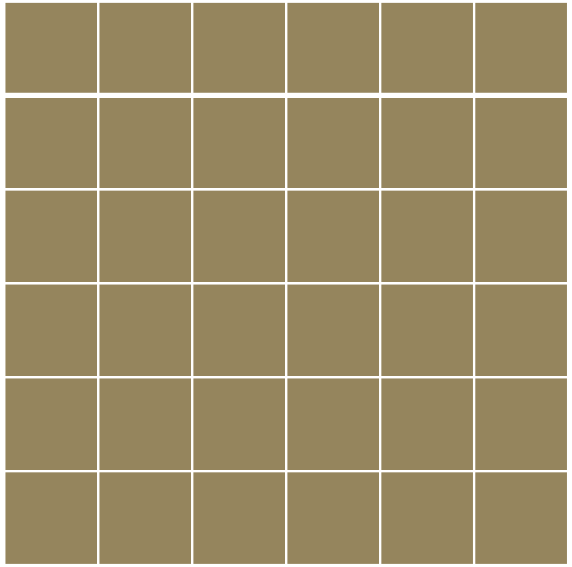
No primeiro trabalho de Inteligência Artificial escolhemos o jogo **Gekitai** ou **Push Away**

O jogo consiste em uma board de 6x6 com 8 peças para cada jogador e o objetivo é ganhar ao oponente fazendo 3 em linha ou tendo 8 peças em campo da sua cor.



Sempre que colocada uma peça as peças a volta em todas as direções no raio de 1 são repelida, caso haja duas peças seguidas ou seja uma peça numa direção n de raio 1 e outra peça na mesma direção n de raio 2 essa peça não é repelida. Podemos observar o comportamento nas figuras ao lado. Colocamos a peça no espaço (1,1) e ela repulsa a peça no espaço (2,1) e não repulsa a que está no espaço (1,2). Por causa das razões explicadas.

Matriz [6][6]

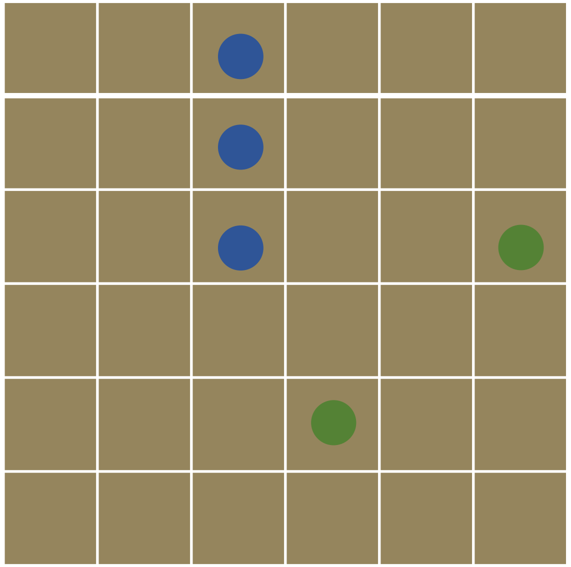


Int Player1 =8
Int Player2 =8

Representação de estado: O estado do jogo será representado por uma matriz [6][6] e dois números inteiros.

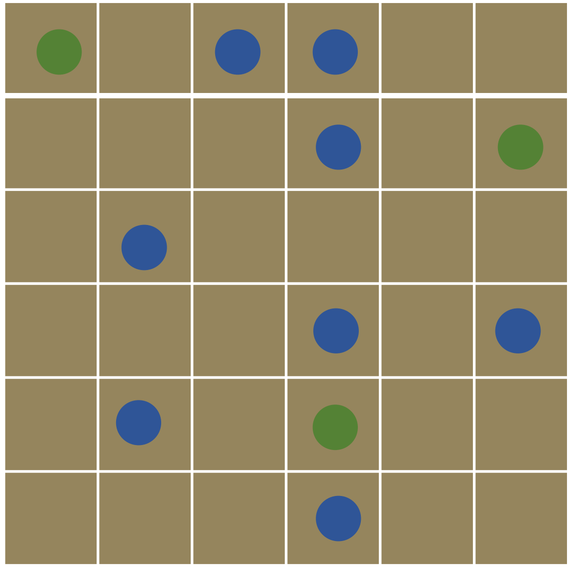
Objetivo: O objetivo é colocar 3 em linhas ou ter 8 peças no Campo como mostra nas figuras

Matriz [6][6]



Int Player1 =8
Int Player2 =8

Matriz [6][6]



Int Player1 =8
Int Player2 =8

Operadores: Neste jogo o único operador que existe é colocar a peça em uma coordenada (X,Y).

Pré condições: A coordenada (X,Y) pertencer a board e não conter nenhuma peça.

Efeitos: Quando colocada a peça, todos os quadrados que contem uma peça que distam 1 a origem são repelidos com exceção dos quadrados que contem também uma peça que distam a 2 na mesma direção.

```
If PieceExist(x+1,y)==True && PieceExist(x+2,y)==False{
        String aux = DeletePiece(x+1,y);
        colocarPiece(x+2,y);
}
```

```
If PieceExist(x,y+1)==True && PieceExist(x,y+2)==False{
        String aux = DeletePiece(x,y+1);
        colocarPiece(x,y+2);
}
```

```
If PieceExist(x-1,y)==True && PieceExist(x-2,y)==False{
        String aux = DeletePiece(x-1,y);
        colocarPiece(x-2,y);
}
```

```
If PieceExist(x,y-1)==True && PieceExist(x,y-2)==False {
        String aux = DeletePiece(x,y-1);
        colocarPiece(x,y-2);
}
```

```
If PieceExist(x+1,y+1)==True && PieceExist(x+2,y+2)==False{
        String aux = DeletePiece(x+1,y+1);
        colocarPiece(x+2,y+2);
}
```

```
If PieceExist(x-1,y+1)==True && PieceExist(x-2,y+2)==False{
        String aux = DeletePiece(x-1,y+1);
        colocarPiece(x-2,y+2);
}
```

```
If PieceExist(x-1,y-1)==True && PieceExist(x-2,y-2)==False{
        String aux = DeletePiece(x-1,y-1);
        colocarPiece(x-2,y-2);
}
```

```
If PieceExist(x+1,y-1)==True && PieceExist(x+2,y-2)==False{
        String aux = DeletePiece(x+1,y-1);
        colocarPiece(x+2,y-2);
}
```

Heurística: Pensamos em vários tipos de heurística, numero de peças na board mais numero de duas peças consecutivas na board ou numero de peças no centro do tabuleiro.

Estrutura de Dados:

```
Class Game{
    String board[][];
    int player1;
    int player2;
    Game(){
        board = New String[6][6];
        player1 = 8;
        player2 = 8;
        for(int i=0;i<6;i++){
            for(int j=0;j<6;j++){
                board[i][j]=“ ”;
            }
        }
    }
```

Função de avaliação: esta função verifica que o numero de peças do jogador chega a 0 e se algum player tem 3 em linha no tabuleiro. Se ambas as condições forem verdade e derem pessoas diferentes da Draw se não em qualquer uma das condições retorna o vencedor.

```
public String toString(){}
public String[][] getBoard(){}
public int getPlayer1(){}
public int getPlayer1(){}
public boolean pieceExist(int x,int y){}
```

```
public void putPiece(int player,int x,
int y){}
public void deletePiece(int x,int y){}
public void atualizar(int x, int y){}
public int verifyWinner(){}
```

Approach: Usamos duas heurísticas para medir a condição de vitoria fora o facto de ser terminal o numero de peças duplas iguais seguidas e o numero total de peças com valores diferentes.

```
public int numberPiecesDouble(){
    return numberPiecesDoubleCalc(1)-numberPiecesDoubleCalc(2);
  }

public int valuePieceHeuristic(){
    return valuePieceHeuristicCalc(1)-valuePieceHeuristicCalc(2);
  }
public void setHeuristic(){
    if(terminal!=0){
      if(terminal==2)
        heuristica = 100;
      else
        heuristica = -100;
    }
    heuristica += valuePieceHeuristic()+ 2 * numberPiecesDouble();
  }
```

Approach (cont)

Para Calcular o numero de peças duplas

```
public int numberPiecesDoubleCalc(int jogador){
  int aux=0;
  if(jogador==2){
    for(int i=0;i<6;i++){
      for(int j=0;j<6;j++){
        if(!state.board[i][j].equals("x")&&!state.board[i][j].equals(" ")){
          if(j+1!=6 &&state.board[i][j].equals(state.board[i][j+1])){
            aux++;
          }
          if(j+1!=6 && i+1!=6 &&state.board[i][j].equals(state.board[i+1][j+1])){
            aux++;
          }
          if(j+1!=6 && i-1!=-1 &&state.board[i][j].equals(state.board[i-1][j+1])){
            aux++;
          }
          if(i+1!=6 && state.board[i][j].equals(state.board[i+1][j])){
            aux++;
          }
        }
      }
    }
  }
  else{
      for(int i=0;i<6;i++){
        for(int j=0;j<6;j++){
          if(!state.board[i][j].equals("o")&&!state.board[i][j].equals(" ")){
            if(j+1!=6 && state.board[i][j].equals(state.board[i][j+1])){
              aux++;
            }
            if(j+1!=6 && i+1!=6 && state.board[i][j].equals(state.board[i+1][j+1])){
              aux++;
            }
            if(j+1!=6 && i-1!=-1 &&state.board[i][j].equals(state.board[i-1][j+1])){
              aux++;
            }
            if(i+1!=6 && state.board[i][j].equals(state.board[i+1][j])){
              aux++;
            }
          }
        }
      }
    }
    return aux;
  }
```

Approach (cont)

Para Calcular o numero de peças no tabuleiro com diferentes valores.

```
public int valuePieceHeuristicCalc(int player){
    int value[][] = {{1,1,1,1,1,1},{1,2,2,2,2,1},{1,2,3,3,2,1},{1,2,3,3,2,1},{1,2,2,2,2,1},{1,1,1,1,1,1}};
    int aux=0;
    for(int i=0;i<6;i++){
      for(int j=0;j<6;j++){
        if(player==1){
          if(state.board[i][j].equals("o")){
            aux+=value[i][j];
          }
        }
        else{
          if(state.board[i][j].equals("x")){
            aux+=value[i][j];
          }
        }
      }
    }
    return aux;
  }
```

Implemented algorithms: O algoritmo escolhido para a realização deste trabalho foi o MinMax, tínhamos planos para fazer o MinMax com os cortes Alpha e Beta mas não houve tempo para realizar isso

```
class MinMaxTree{
  Game state;
  List<MinMaxTree> nodeList;
  int heuristica;
  int terminal;
  MinMaxTree(Game dadState){
    state = new
Game(dadState.getBoard(),dadState.getPlayer1(),dadState.getPlayer2());
    nodeList = new ArrayList<>();
    terminal = 0;
  }
public void expande(int depth){}
public void setHeuristicMax(){}
public void setHeuristicMin(){}
public int numberPiecesDouble(){}
public int numberPiecesDoubleCalc(int jogador){}
public void setHeuristic(){}
public int valuePieceHeuristicCalc(int player){}
public int valuePieceHeuristic(){}
```

Quando criada a arvore MinMaxTree é chamado o expande que expande os nós até uma profundidade máxima dada e de seguida atribui os valores da heurística aos seus respetivos nós chamando o setHeuristicMin() no caso do player1 ou SetHeuristicMax() no caso do player2

Heurísticas:

Número de Peças: Calcula número das suas peças contra o número de peças do adversário em campo;

Duas Peças: Verifica se tem duas peças consecutivas;

Valor das Peças: As peças no centro do tabuleiro têm mais valor;

A = Número de Peças + 2 * Duas Peças

B = Valor de Peças + 2 * Duas Peças

Resultado: Percentagem de vitórias de cada bot numa amostra de 10 jogos consecutivos entre cada um deles;

Bot 1	Bot 2	Resultado (% vitorias Bot 1 - % vitórias Bot 2)
Número de Peças, Profundidade 1	**Duas Peças, Profundidade 1**	100% - 0%
Número de Peças, Profundidade 1	**Duas Peças, Profundidade 2**	60% - 40%
Número de Peças, Profundidade 1	**Duas Peças, Profundidade 3**	20% - 80%
Número de Peças, Profundidade 2	**Duas Peças, Profundidade 3**	50% - 50%
Número de Peças, Profundidade 3	**Duas Peças, Profundidade 3**	100% - 0%
Número de Peças, Profundidade 1	**Valor de Peças, Profundidade 1**	0% - 100%
Número de Peças, Profundidade 2	**Valor de Peças, Profundidade 1**	30% - 70%
Número de Peças, Profundidade 3	**Valor de Peças, Profundidade 1**	10% - 90%
Número de Peças, Profundidade 3	**Valor de Peças, Profundidade 3**	0% - 100%
A, Profundidade 1	**Valor de Peças, Profundidade 1**	0% - 100%
A, Profundidade 3	**Valor de Peças, Profundidade 3**	0% - 100%
A, Profundidade 1	**B, Profundidade 1**	10% - 90%
A, Profundidade 3	**B, Profundidade 3**	0% - 100%
A, Profundidade 1	**Número de Peças, Profundidade 1**	20% - 80%
B, Profundidade 1	**Valor de Peças, Profundidade 1**	0% - 100%
B, Profundidade 1	**Número de Peças, Profundidade 1**	80% - 20%
B, Profundidade 3	**Número de Peças, Profundidade 3**	60% - 40%

Conclusão

Para este jogo em questão parece-me que o MinMax é um bom algoritmo de decisão porém sem o cortes alfa beta o espaço é demasiado grande para um depth superior a 3 o que possivelmente torna o jogo mais fácil de ser ganho no nosso caso. Também decidimos pesquisar sobre o algoritmo de Monte Carlo porem pareceu complexo para o tempo que tínhamos e decidimos implementar o MinMax dado na aula.

Para este algoritmo é importante ter uma heurística bem definida assim como os estados terminais.

Apoio na Pesquisa

Para obter informação sobre o trabalho usamos os seguintes linkes:
https://boardgamegeek.com/boardgame/295449/gekitai - link da descrição do jogo
https://chessicals.com/games/gekitai/ - link de um jogo online com o mesmo objetivo