**Universidade Federal de Santa Catarina**

**INE5430 – Inteligência Artificial**

**Prof. Elder Rizzon Santos**

**Ciências da Computação**

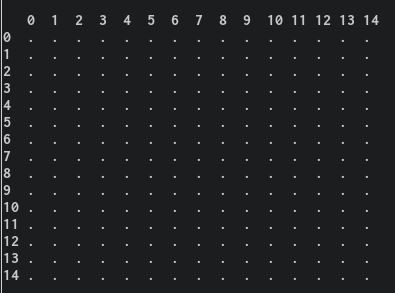
**Trabalho sobre Métodos de busca**

**Willian de Souza – 11100901**

1. **Introdução**

Para este trabalho, foi usada a linguagem de programação python para construir a lógica do jogo.

A Interface de acesso ao usuário foi produzida de forma simples baseada no console [*Figura 1*].

  
Figura 1. Tabuleiro do jogo

Como visto na *Figura 1,* a representação do tabuleiro foi feita através de matriz, onde o “.” representa um campo em branco, ou seja, um campo ainda não consumido pelo jogo. O jogo conta com duas representações de peças, peça *preta* representada pela letra “x” e a peça *branca* representada pela letra “o”. O fonte da representação segue abaixo.

**class** Piece:

*"""Piece representation*

*"""*

**"""Indicate the position in board does not played yet"""**

NONE = **'.'**

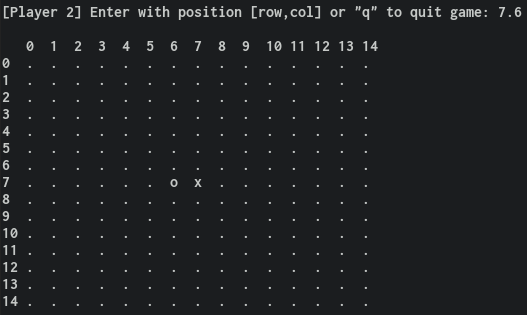
**"""Indicate a BLACK piece for game"""**

BLACK = **'x'**

**"""Indicate a WHITE piece for game"""**

WHITE = **'o'**

A entrada dos dados para o jogo é no formato “linha, coluna” como segue na *Figura 2*, onde mostra também as posições jogadas pelo jogador 1 e jogador 2.

  
Figura 2. Estado do tabuleiro apó a jogada do Player 1 seguida pelo Player 2, na posição [7,6]

1. **Implementação**
   1. **Player**

Para representar os Players (Jogadores), foi usada uma representação abstrata chamada *AIPlayer* como segue:

**class** AIPlayer:

*""" Abstract AI players.*

*To construct an AI player:*

*Construct an instance (of its subclass) with the game Board*

*"""*

**def** \_\_init\_\_(self, board, piece):

self.\_board = board

self.my\_piece = piece

self.opponent = Piece.WHITE **if** piece == Piece.BLACK **else** Piece.BLACK

**def** play(self, row, col):

**raise** NotImplemented

* 1. **Human Player**

Humano é um tipo de player, essa é a representação mais simples, ela é manipulada através das entradas de uma pessoa pelo console como visualizado na *Figura 2*.

Um Humano herda de *AIPlayer* define e sobrescreve a função play, que recebe a linha e a coluna e insere a peça correspondente ao jogador no tabuleiro. Essa função também é responsável por gerenciar se a jogada pode ser efetuada ou não. De maneira simples pode-se exemplificar da seguinte forma, se um *jogador 1* jogou sua peça “*x”* na linha 7 e coluna 7 e o *jogador 2* tentar jogar sua peça “*o”* na mesma posição, temos uma situação de conflito e a regra do jogo diz que não podemos sobrescrever uma jogada colocando outra peça na mesma posição em que uma peça já se encontra, dessa forma é lançada uma *exception* (exceção), para ser tratada no nível de aplicação.

**class** HumanPlayer(AIPlayer):

*""" Human Player*

*"""*

**def** \_\_init\_\_(self, board, piece, first=**True**):

super(HumanPlayer, self).\_\_init\_\_(board, piece)

self.first = **not** first

**def** play(self, row, col):

**if** self.\_board.get\_piece(row, col) != Piece.NONE:

**raise** OverwritePositionException

**return** self.\_board.play\_piece(self.my\_piece, row, col)

**def** \_\_repr\_\_(self):

player\_number = int(self.first) + 1

**return 'Player {}'**.format(player\_number)

* 1. **Machine Player**

Machine também é uma das abstrações de um player, de forma diferente do modelo *Human* nosso modelo formatado através da classe *Machine* realiza jogadas de forma autônoma, usando como auxílio um algoritmo chamado “*minimax*” em conjunto com heurísticas. O minimax é o responsável por calcular e determinar a próxima jogada a partir do estado atual.

**class** MachinePlayer(AIPlayer):

*""" Machine Player*

*"""*

**def** \_\_init\_\_(self, board, piece, level=2):

self.first\_move = **True**

self.level = level

super(MachinePlayer, self).\_\_init\_\_(board, piece)

**def** play(self, square=**None**):

*""" Move the piece of player*

**:param** *square: is the position to move, in this case not used*

**:return***: winner piece if has one, Piece.None otherwise*

*"""*

**if** self.first\_move:

self.first\_move = **False**

middle = int(len(self.\_board) / 2)

square = Square(middle, middle)

**if** self.\_board.get\_piece(square) != Piece.NONE:

square = Square(middle - 1, middle - 1)

**return** self.\_board.take\_up\_space(self.my\_piece, square)

\_, square = self.minimax\_pruning(self.level)

**if** self.\_board.get\_piece(square) != Piece.NONE:

**raise** OverwritePositionException

**return** self.\_board.take\_up\_space(self.my\_piece, square)

**def** \_minimax(self, level, player):

*""" The simple form MiniMax algorithm*

**:param** *level: level to down in the tree*

**:param** *player: current player*

**:return***: tuple(score, position) score is the best score found, and*

*position is the best move*

*"""*

current\_score = 0

best\_movement = Square(-1, -1)

best\_score = -sys.maxsize **if** (player == self.my\_piece) **else** sys.maxsize

**if** level == 0:

**return** (self.\_board.current\_score, best\_movement)

**for** row **in** range(len(self.\_board)):

**for** col **in** range(len(self.\_board)):

current\_movement = Square(row, col)

**if** self.\_board.get\_piece(current\_movement) != Piece.NONE:

**continue**

previous\_score = self.\_board.current\_score

self.\_board.take\_up\_space(player, current\_movement)

**if** player == self.my\_piece: *# me*

current\_score, \_ = self.\_minimax(level - 1, self.opponent)

**if** current\_score > best\_score:

best\_score = current\_score

best\_movement = current\_movement

**else**: *# opponent*

current\_score, \_ = self.\_minimax(level - 1, self.my\_piece)

**if** current\_score < best\_score:

best\_score = current\_score

best\_movement = current\_movement

*# undo move*

self.\_board.restore\_move(current\_movement, previous\_score)

**return** (best\_score, best\_movement)

No minimax como informado na definição:

“Em teoria da decisão, o minimax (ou minimax) é um método para minimizar a perda máxima possível.”

No algoritmo Minimax temos dois parâmetros *level* e *player*. O parâmetro *level* indica quantos níveis da árvore devemos descer para analisar melhor jogada e o parâmetro *player* informa o jogador que está efetuando a jogada. Temos dois possíveis tratamentos, um para cada jogador. Na visão da Maquina precisamos maximar as chances para concluir nossa sequência de jogadas em vitória, por outro lado devemos minimizar as chances do adversário ganhar.

No Minimax percorremos o laço na forma de linha X coluna, ou seja, o laço é percorrido da esquerda linha por linha, simulando as jogadas e guardando a melhor jogada (aquela que gera mais pontos em uma jogada). Esse algoritmo é recursivo e tem como retorno a posição da melhor jogada e os pontos consequentes.

Na *Figura 3*, temos um tabuleiro antes da jogada e na *Figura 4* temos a consequência da jogada. Como no algoritmo temo a busca da melhor jogada feita da esquerda para a direita começando pela primeira linha, temos que o maior pontuação retornada neste caso e o estado “filho” que resulta na jogada feita na posição 6x6.

→

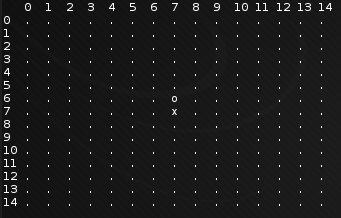
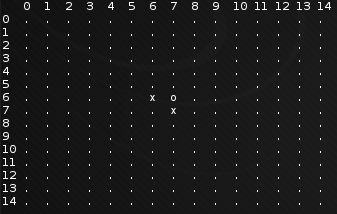
Figura 3: Estado atual

Figura 4: Próximo estado, obtido a partir do estado na Figura 3

O algoritmo de Minimax possui uma outra forma de implementação usando podas com o auxílio das variáveis *alpha* e *beta.*

Segue o fonte abaixo.

**def** \_minimax\_pruning(self, level, player, alpha, beta):

*""" MiniMax algorithm with alpha and beta pruning*

**:param** *level: level to down in the tree*

**:param** *player: current player*

**:param** *alpha: alpha score*

**:param** *beta: beta score*

**:return***: tuple(score, position) score is the best score found, and*

*position is the best move*

*"""*

score = 0

best\_movement = Square(-1, -1)

**if** level == 0:

**return** (self.\_board.current\_score, best\_movement)

**for** row **in** range(len(self.\_board)):

**for** col **in** range(len(self.\_board)):

current\_movement = Square(row, col)

**if** self.\_board.get\_piece(current\_movement) != Piece.NONE:

**continue**

previous\_score = self.\_board.current\_score

self.\_board.take\_up\_space(player, current\_movement)

**if** player == self.my\_piece: *# me*

score, \_ = self.\_minimax\_pruning(level - 1, self.opponent, alpha, beta)

**if** score > alpha:

alpha = score

best\_movement = current\_movement

**else**: *# opponent*

score, \_ = self.\_minimax\_pruning(level - 1, self.my\_piece, alpha, beta)

**if** score < beta:

beta = score

best\_movement = current\_movement

*# undo move*

self.\_board.restore\_move(current\_movement, previous\_score)

**if** (alpha >= beta):

**break**

score\_return = alpha **if** player == self.my\_piece **else** beta

**return** (score\_return, best\_movement)

Nessa versão o algoritmo é chamado de “*Minimax com poda de Alpha e Beta”*. A implementação é feita de forma semelhante ao algoritmo simples de Minimax apresentado anteriormente, a diferença é que usamos os valores de *alpha* e *beta* para evitar que o algoritmo continue procurando por mais possibilidades, pois as melhores possibilidade possíveis já estão definidas em *alpha* e *beta*.

Dessa maneira esse algoritmo se mostra muito superior ao anterior, pelo fato de não precisar esgotar todas as possibilidades para encontrar a melhor jogada.

Na *Figura 5*, temos testes realizados sobre as mesmas condições com os dois algoritmos, nesses testes podemos visualizar a grande eficiência do algoritmo de Minimax com podas de *alpha* e *beta*.

Figura 5: Benchmark dos algoritmos de Minimax

* 1. **Border**

Border neste momento talvez seja a principal implementação, ou seja, é nela que se encontra atualmente a maior.

O tabuleiro recebe como parâmetros dois números, o primeiro “size” usado para construir a tabela, que é uma matriz quadrada (size X size) e o segundo “sequence\_victory” é usado para determinar a sequência de peças iguais que constitui em uma vitória.

Nesta classe temos a API necessária para que outras classes possam manipular o tabuleiro com facilidade, dentro dela está a função play\_piece mostrada abaixo.

**def** play\_piece(self, piece, row, col):

self.\_table[row, col] = piece

winner = self.has\_winner(piece, row, col)

**if** (self.\_table == **'.'**).sum() == 0 **and** winner == Piece.NONE:

**raise** NotBlankSpaceException

**return** winner

Esta função é responsável por gerenciar o que pode ocorrer durante uma jogada, ou seja, após a jogada é verificado se possui algum vencedor na rodada atual, se possuir o mesmo é retornado, porém ela também é responsável por verificar se o jogo terminou sem ganhador, através da *exception* NotBlankSpaceException lançada para o nível de cima da aplicação. Essa *exception* pode ser interpretada como uma caso de empate, onde acabam as casas (posições) possíveis para o jogo.

A maior implementação desta classe se dá por conta de uma busca no tabuleiro, para encontrar um vitorioso, essa busca é feita atravé da função has\_winner, que faz a procura do vencedor pelas diferentes opções que se pode alcançar a vitória.

**def** has\_winner(self, piece, row, col):

**if** self.\_search\_line(piece, row, col):

**""" Faz verificação por linha**

**se ocorrer uma vitória na linha atual**

**escreve a peça ganhadora e retorna verdadeiro**

**"""**

**return** piece

**if** self.\_search\_column(piece, row, col):

**""" Faz verificação por linha**

**se ocorrer uma vitória na linha atual**

**escreve a peça ganhadora e retorna verdadeiro**

**"""**

**return** piece

**if** self.\_search\_diagonal(piece, row, col):

**""" Faz verificação da diagonal no sentido da diagonal principal**

**se ocorrer uma vitória na diagonal atual**

**escreve a peça ganhadora e retorna verdadeiro**

**"""**

**return** piece

**if** self.\_search\_opposite\_diagonal(piece, row, col):

**""" Faz verificação da diagonal no sentido da diagonal secundária**

**se ocorrer uma vitória na diagonal atual**

**escreve a peça ganhadora e retorna verdadeiro**

**"""**

**return** piece

**return** Piece.NONE

Essa função usa outras quatro funções que auxiliam na busca da vitória \_search\_line, \_search\_column, \_search\_diagonal e \_search\_opposite\_diagonal. E essas funções utilizam a função chamada \_victory\_match, que gera a forma vencedora para fazer a avaliação das posições (ex. “*xxxxx”* que é uma sequência válida para a vitória do jogador com a peça “*x*”) segue a implementação.

**def** \_victory\_match(self, piece):

*""" A generator to match victory*

**:param** *piece: for check the victory sequence*

**:return***: the sequence for victory*

*"""*

**return** piece \* self.\_sequence\_victory

As funções de busca da vitória pela linha e pela coluna são um pouco semelhantes. A busca por linha pega através da posição da jogada atual, cinco colunas anteriores da atual e cinco colunas após, respeitante os limites do tabuleiro, transforma essa lista retornada em uma *string* e com o auxílio da função \_victory\_match verifica se a *string* necessária para garantir a vitória está contida na *string* transformada. De formar semelhante a busca por coluna é feita, considerando que as movimentações para a verificação do intervalo são feitas nas linha e da mesma forma a lista retornada é convertida para *string* e comparada com a match\_str.

**def** \_search\_line(self, piece, row, col):

*""" Search has victory in line*

*Check in matrix row if has match value to victory*

*it get the previous five column position from current position played and*

*the next five column position from current and check if has victory*

**:param** *piece: current piece played*

**:param** *row: position in matrix*

**:param** *col: position in matrix*

**:return***: if has a victory in current line, True*

*otherwise is False*

*"""*

match\_str = self.\_victory\_match(piece)

start\_col = 0 **if** (col - self.\_sequence\_victory) < 0 **else** (col - self.\_sequence\_victory)

size = len(self.\_table)

end\_col = size **if** (col + self.\_sequence\_victory + 1) > size **else** (col + self.\_sequence\_victory + 1)

**return** match\_str **in ''**.join(self.\_table[row, start\_col:end\_col])

**def** \_search\_column(self, piece, row, col):

*""" Search has victory in line*

*Check in matrix column if has match value to victory*

*it get the previous five row position from current position played and*

*the next five row position from current and check if has victory*

**:param** *piece: current piece played*

**:param** *row: position in matrix*

**:param** *col: position in matrix*

**:return***: if has a victory in current row, True*

*otherwise is False*

*"""*

match\_str = self.\_victory\_match(piece)

start\_row = 0 **if** (row - self.\_sequence\_victory) < 0 **else** (row - self.\_sequence\_victory)

size = len(self.\_table)

end\_row = size **if** (row + self.\_sequence\_victory + 1) > size **else** (row + self.\_sequence\_victory + 1)

**return** match\_str **in ''**.join(self.\_table[start\_row:end\_row, col])

A função de busca pela diagonal (que é no mesmo sentido da diagonal principal), foi feita com a ajuda de uma função da biblioteca matemática *numpy*. A ideia é gerar offset que a diagonal tem em relação à diagonal principal e com a ajuda da função *diagonal* da biblioteca, pegamos a diagonal solicitada no formato de lista e da mesma forma convertemos para *string* e então comparamos com a match\_str.

**def** \_search\_diagonal(self, piece, row, col):

*""" Search has victory by diagonal*

*Check in matrix the diagonal if has match value to victory*

*it get the previous five diagonal (row x col) position from current position played and*

*the next five diagonal (row x col) position from current and check if has victory*

**:param** *piece: current piece played*

**:param** *row: position in matrix*

**:param** *col: position in matrix*

**:return***: if has a victory in current diagonal, True*

*otherwise is False*

*"""*

match\_str = self.\_victory\_match(piece)

offset = col - row

diagonal = np.diag(self.\_table, k=offset).tolist()

diagonal\_formatted = **''**.join(diagonal)

**return** match\_str **in** diagonal\_formatted

Já na função de busca pela diagonal oposta, a ideia foi girar o tabuleiro 90 graus no sentido anti-horário, fazer a conversão das posições (de 15. 14. …. 1. 0 para 0. 1. …. 14. 15) que ficaram correspondentes às linhas. Essa alteração é apenas no formato de representação dos dados, não afeta a tabela original e facilita pegarmos a diagonal oposta usando os métodos usados na captura da diagonal principal.

**def** \_search\_opposite\_diagonal(self, piece, row, col):

*""" Search has victory by diagonal*

*Check in matrix the diagonal if has match value to victory*

*it get the previous five diagonal (row x col) position from current position played and*

*the next five diagonal (row x col) position from current and check if has victory*

**:param** *piece: current piece played*

**:param** *row: position in matrix*

**:param** *col: position in matrix*

**:return***: if has a victory in current diagonal, True*

*otherwise is False*

*"""*

new\_col = row

size = len(self.\_table)

new\_row = (size - 1) - col

match\_str = self.\_victory\_match(piece)

offset = new\_col - new\_row

column\_inverted = self.\_table[:, ::-1]

transposed = column\_inverted.transpose()

diagonal = np.diag(transposed, k=offset).tolist()

diagonal\_formatted = **''**.join(diagonal)

**return** match\_str **in** diagonal\_formatted

1. **Heurística**

Nossa heurística é calculada em cada jogada, atualizando o *score point* (pontos apresentados no tabuleiro). Para calcular, é pega a posição da última jogada da faz a verificação em todos os sentidos possíveis de vitória, ou seja, pela coluna, pela linha ou pelas diagonais. Dada a posição atual, pega-se uma sublista da posição atual -5 até a posição atual +5, essa sublista é passada para a função *heristic\_move\_score* para calcular o novo valor do tabuleiro baseado na jogada atual.

*SCORE\_POINT* é uma lista com as pontuações, a posição de cada ponto corresponde a sequência encontrada na lista.

***SCORE\_POINT = [***

0,

1,

100,

10000,

1000000,

50000000000

]

***def*** *heuristic\_move\_score(self, piece, line):*

*""" Calculate the current move*

*Calculate the score of current move in line list*

*This calc is done searching from maximum point until minimum point*

**:param** *piece: current piece to calculate score*

**:param** *line: list to calcilate*

*"""*

score\_factor = 1 **if** piece == Piece.BLACK **else** -1

opponent = Piece.WHITE **if** piece == Piece.BLACK **else** Piece.BLACK

line = **''**.join(line)

**for** i **in** range(self.\_sequence\_victory, 0, -1):

**""" A generator to match sequence to search score**

**"""**

match = piece \* i

**if** match **in** line:

bad\_move = opponent + match + opponent

**if** bad\_move **not in** line:

self.score += (score\_factor \* SCORE\_POINT[i])

**return**

Essa função heurística possui uma falha, a heurística calcula a sequência de maneira correta, porém quando sequência é bloqueada a função deveria desconsiderar o valor da sequência, ou seja, quando houver um bloqueio a sequência deve ser subtraída dos pontos do tabuleiro.