**Tic-Tac-Toe**

Tic-Tac-Toe is a game played on a grid that's 3 squares by 3 squares. The first player to get 3 of marks in a row (up, down, across, or diagonally) is the winner. With 255,168 possible scenarios excluding symmetry. The first player wins 131,184 of these and the second player wins 77,904 games and remaining 46080 are drawn.

**AI approach**

In an AI approach. Unlike human players that make our decisions from previous moves, it is merely a tree search problem where every AI’s move is based on how the final game looks like. The AI will traverse maximum of 9! scenarios (more placed mark, less scenarios hold) by assuming humans’ moves on every empty cell, and it will then try every remaining empty cell as response until the game ends, that is, directed by the “Minimax Algorithm”.

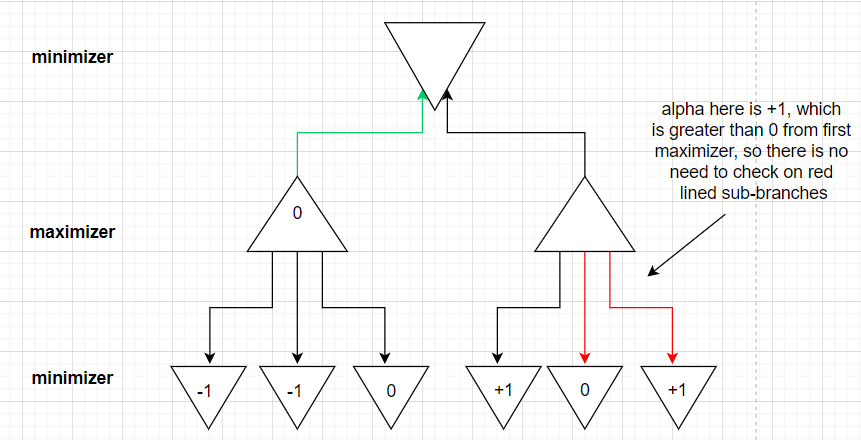
**Minimax algorithm**

After having massive, completed games on the bottom leaf node, the algorithm will then score each game by its result such as +1 for the game where “X” wins, “-1” for “O”, and 0 for draws. Depending on AI’s role as “X” or “O”, it favors the results which give the corresponding winning score, and if there isn’t one, it then favors the draw which is score 0. In this case, “X” looks for value as positive as possible which is so called maximizer and “O” is the minimizer.

When AI, let’s say “X”, have selected all favored results from bottom leaves. It then brings them to the upper level. Since the players switch turns, the algorithm then plays in the perspective of “O” and pick the lowest score from the games that brought by “X”, and this is because it assumes human player always make optimal move just like itself does. This continues all the way up to the current stage where human player is waiting for AI’s move, it then selects the highest score scenarios from a list of scenarios generated from the bottom-up process and guarantee it’s the best move it could make.

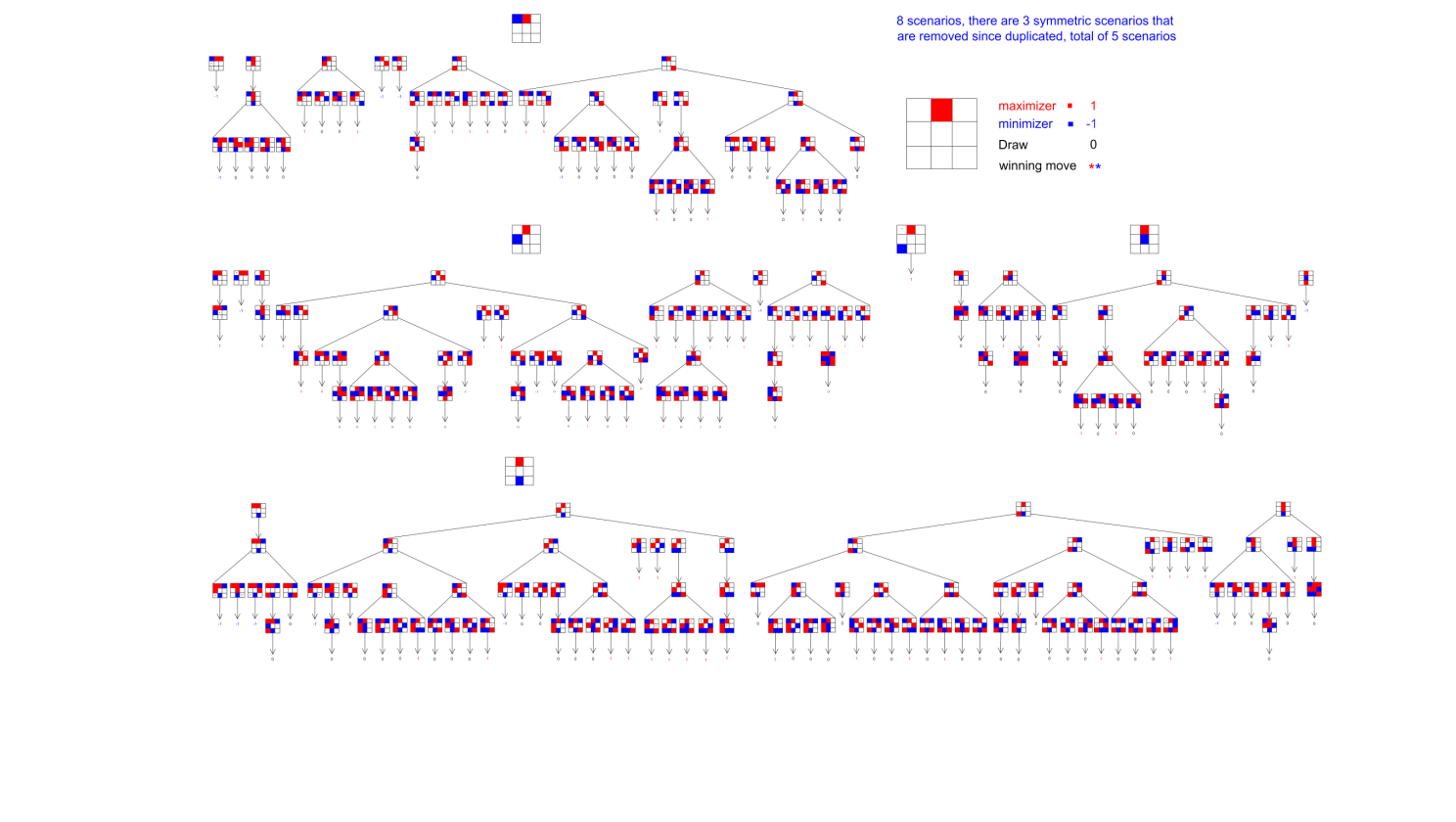
**Alpha-Beta Pruning**

The Alpha-Beta Pruning is just an advanced extension of Minimax. Imagine a scenario where There are 2 branches in a maximizer level and its upper and lower levels are minimizers. When the first branch of maximizer have a value of “0”, second maximizer branch gets “+1” from first of its sub-branches, since we know the upper minimizer will pick the lowest score, and the second maximizer will pick that “+1” which is greater than first maximizer who already has “0”, so it is pointless to go over other sub-branches in second maximizer, here is a [simulator](https://pascscha.ch/info2/abTreePractice/).

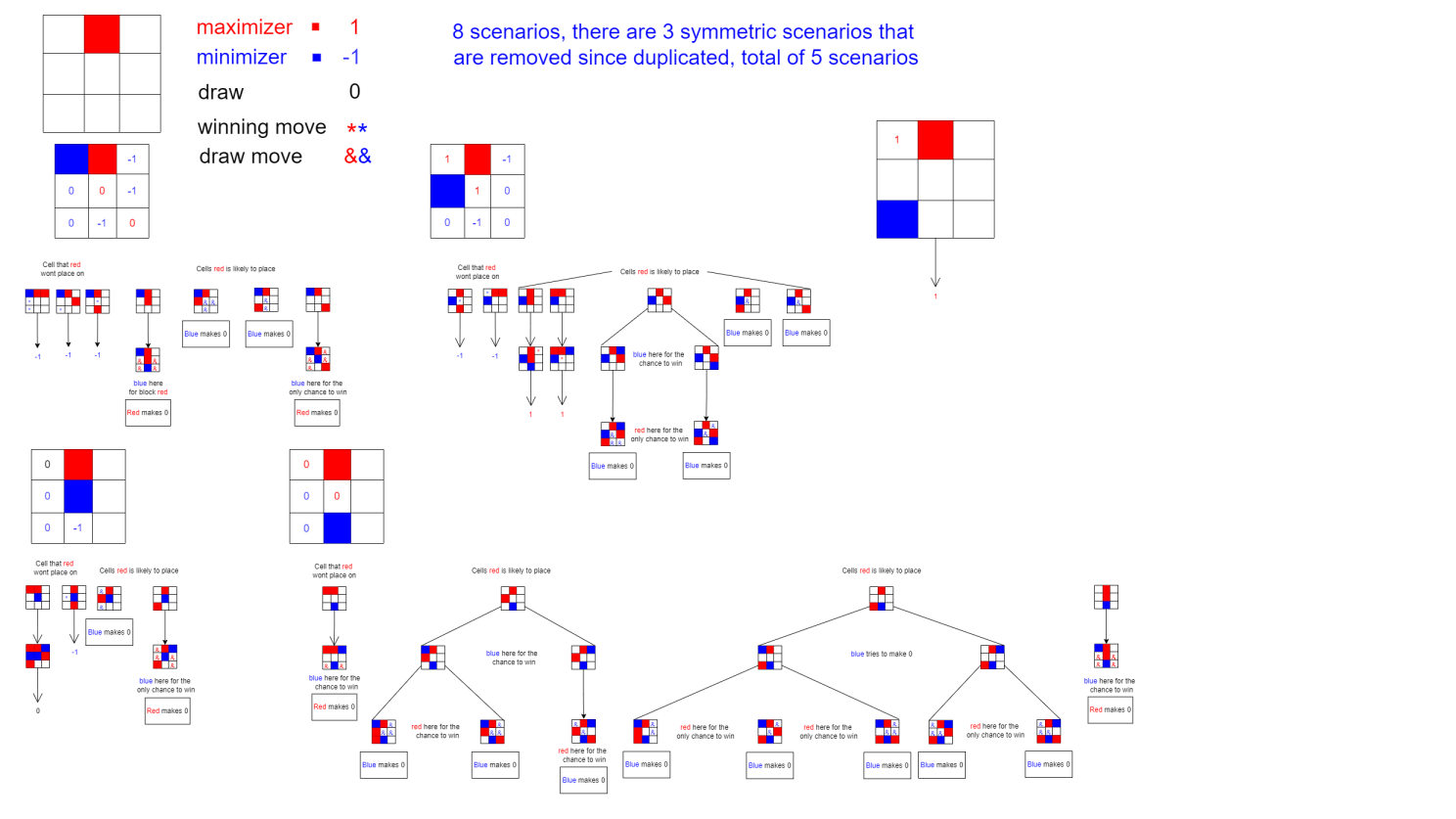


### Work through

To understand tic-Tac-Toe, it's better to draw diagrams to simulate all these scenarios. The first diagram show below is a half-intelligent play through where we assume that 2 players will place randomly but sensitive to the 2 in a line condition (block the obvious loss). For the sake of space, I assume that the first move is always placed at top middle.

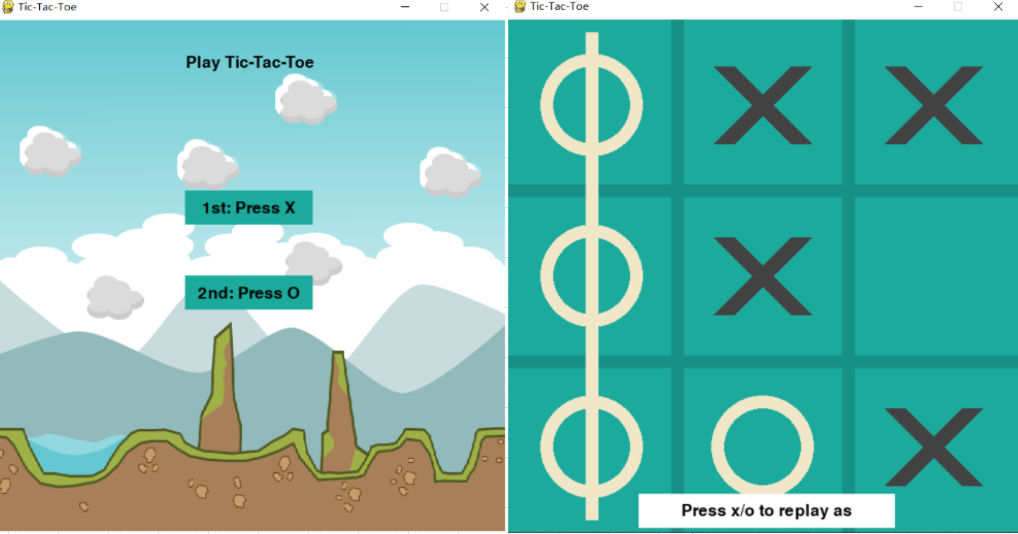
**[[](https://app.diagrams.net/#G1XnTE6ORzHG7sZ7VAyTRkdm6tmHoZMwic)](https://app.diagrams.net/" \l "G1XnTE6ORzHG7sZ7VAyTRkdm6tmHoZMwic" \t "_blank)**

From the perspective of AI, since it will always make optimal moves, so it's not hard to see that if two AI play against each other, it will always yield a draw, and the diagram show below illustrate the scenarios I summarized in short (include but not limited to)

**[[](https://app.diagrams.net/#G1NiaA4maqkTzfcJw0emPAKvfnxGSdkuTM)](https://app.diagrams.net/" \l "G1NiaA4maqkTzfcJw0emPAKvfnxGSdkuTM" \t "_blank)**

### GUI Implementation

The GUI is made by using python.pygame library, where I create a background filled with color, and 2 vertical and horizontal lines to split the board into nine cells. The cover image is from open-source [**background | OpenGameArt.org**](https://opengameart.org/content/background-3) and the pygame design is inspired by [**making tic tac toe in python - YouTube**](https://www.youtube.com/watch?v=pc7XhHxSgrM)



### Algorithm Implementation

While the above game GUI only allow you play normal Tic-Tac-Toe, to achieve the AI functionalities, we then need following functions:

def player(board):

    #    Returns player who has the next turn on a board.

    count\_x=0

    count\_o=0

    for i in board:

        for j in i:

            if(j=="X"):

                count\_x=count\_x+1

            if(j=="O"):

                count\_o=count\_o+1

    return O if count\_x > count\_o else X

def actions(board):

    # get all available empty cells for AI

    action=set()

    for i, row in enumerate(board):

        for j , vall in enumerate(row):

            if(vall==None):

                action.add((i,j))

    return action

def result(board, action):

    i,j=action

    if(board[i][j]!=None):

        raise Exception("Invalid Move ")

    next\_move=player(board)

    deep\_board=deepcopy(board)

    deep\_board[i][j]=next\_move

    return deep\_board

def winner(board):

    # find the winner of the game

    rows=board+get\_diagonal(board) +get\_columns(board)

    for row in rows:

        current\_palyer=row[0]

        if current\_palyer is not None and three\_in\_a\_row(row):

            return current\_palyer

    return None

def terminal(board):

    # see if game is finished, does not return winner

    xx=winner(board)

    if(xx is  not None):

        return True

    if(all(all(j!=None for j in i) for i in board)):

        return True

    return False

def utility(board):

    xx=winner(board)

    if(xx==X):

        return 1

    elif(xx==O):

        return -1

    else:

        return 0

def max\_alpha\_beta\_pruning(board ,alpha,beta,layer):

    if(terminal(board)== True):

        return utility(board) , None

    global chance

    vall=float("-inf")

    best=None

    for action in actions(board):

        min\_val=min\_alpha\_beta\_pruning(result(board ,action), alpha, beta,layer+1)[0]

        if layer == 0:

            chance[action] = min\_val

        if( min\_val > vall):

            best=action

            vall=min\_val

        alpha=max(alpha,vall)

        if (beta <= alpha):

            break

    return vall,best

def min\_alpha\_beta\_pruning(board ,alpha,beta, layer):

    if(terminal(board)== True):

        return utility(board) , None

    global chance

    vall=float("inf")

    best=None

    for action in actions(board):

        max\_val=max\_alpha\_beta\_pruning(result(board ,action), alpha, beta,layer+1)[0]

        if layer == 0:

            chance[action] = max\_val

        if( max\_val < vall):

            best=action

            vall=max\_val

        beta=min(beta,vall)

        if (beta <= alpha):

            break

    return vall,best

def minimax(board):

    if terminal(board):

        return None

    if(player(board)==X):

        return max\_alpha\_beta\_pruning(board ,float("-inf") ,float("inf"), 0)[1]

    elif(player(board) == O):

        return min\_alpha\_beta\_pruning(board , float("-inf"), float("inf"), 0)[1]

    else:

        raise Exception("Error in Caculating Optimal Move")

def show\_AI\_chances(board):

    global chance

    for row in range(3):

        for col in range(3):

            if board[row][col] != None and (row,col) in chance:

                chance.pop((row,col))

    for c in chance:

        print(c, chance[c])

    return chance

For reference, please see [**Play Tic-Tac-Toe with AI | Python | Minimax | Alpha-Beta Pruning | Artificial Intelligence - YouTube**](https://www.youtube.com/watch?v=qPN28mzHC08)

The cool thing is, I create a python dictionary to store the minimax score generated from each AI move (global variable chance in the code), This will then show on the board while playing and interprets whether a cell is good or bad for the AI, and you will find that it's actually performing as designed, which is, looking for a high score as maximizer ("X") and looking for a low score as minimizer ("O").

Now, let's see the video down below and you will know exactly what I'm talking about.

Since the "X" is the maximizer, the number "-1" shown on the board are the move the "X" is trying to avoid, otherwise there would be a chance where "O" can win the game. On the other hand, when AI plays as "O", it looks for the cell with "-1" first and placing on a cell with "+1" will be unwise.

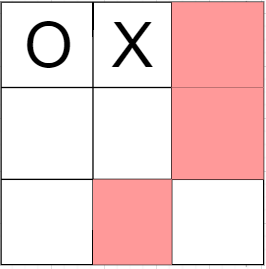
### Summarization

Even though 2 AI will always yield a draw, we human players still have to pay close attention while playing. Here are the rules summarized from this implementation, because of symmetry, there are three starting moves: corner, middle, and center.

**Middle**

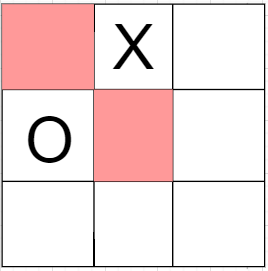
When X first places on the middle, nothing could be determined yet.

1. When O places on the sides of the same row/column, eg:



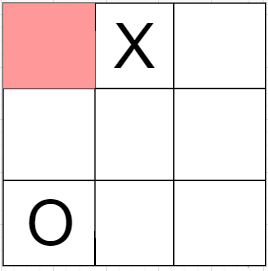
If X responses on these 3 positions marked in pink, X will lose the game.

2. When O places on the sides of adjacent row/column, eg:



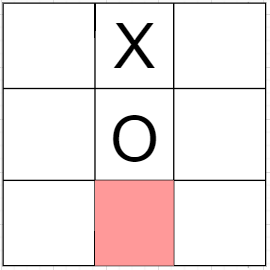
In this case, X has already won the game by placing on any of these positions marked as pink.

3. When O places on the sides of non-adjacent row/column, eg:



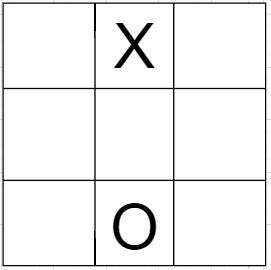
X has already won the game by place on the pink cell.

4. When O places on the center, eg:



X should avoid place on the colored cell which leads to a loss for X.

5. When O places on the middle of non-adjacent row/column, eg:

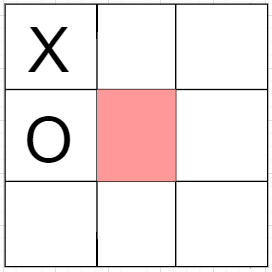


This should come to a draw unless we intentionally ignore where the other player places the remaining marks on. (Since X has the edge to go first, this is suggested move for O to draw the game)

**Corner**

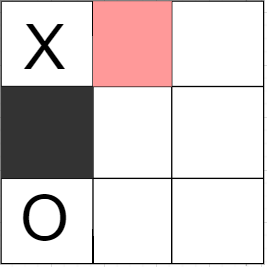
Just like middle, nothing could be determined at first move of X.

1. When O places on the middle of adjacent row/column, eg:



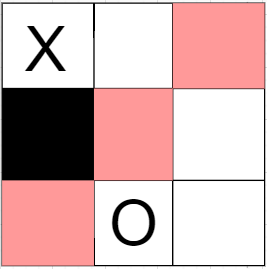
In this case X can easily win the game by placing on the center.

2. When O places on the corner of non-adjacent row/column, eg:



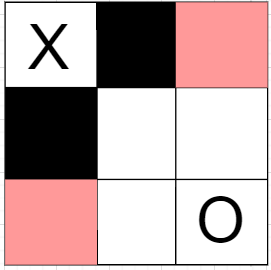
X could play on pink to guarantee winning, or if X places on black, it's a win for O

3. When O places on the middle of non-adjacent row/column, eg:



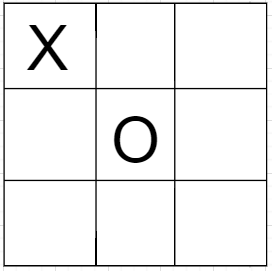
X could win the game directly by playing on pink cells, but also has chance to lose if X accidently places on black cell.

4. When O places on the diagonal corner of non-adjacent row/column, eg:



X can win by placing on pink cells and would lose if place on black.

5. When O places on the center, eg:

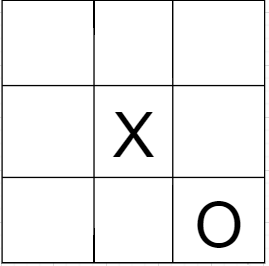


This should come to a draw unless we intentionally ignore where the other player places the remaining marks on. (Since X has the edge to go first, this is suggested move for O to draw the game)

**Center**

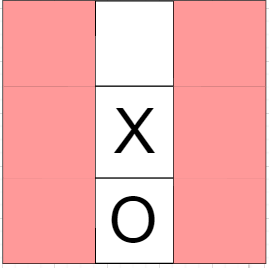
Again, there should be a draw before second move made by O

1. When O place on corner, eg:



This should come to a draw unless we intentionally ignore where the other player places the remaining marks on. (Since X has the edge to go first, this is suggested move for O to draw the game)

2. When O places on middle of adjacent row/column, eg:



X easily wins the game; it could place on anywhere except the middle of the other side.

### Conclusion

Overall, the player goes first has tremendous edge of not losing, which means it can at least lead the game to a draw no matter playing against an AI or a human player.

On the other hand, the player goes second needs to pay attention, for middle, corner and center occupied start, second player will have 3/5, 4/5, 4/5 scenarios to lose the game, especially when playing against an AI who will never make mistakes. But if you play smartly, the game could always be drawn.