COMP 2211 Exploring Artificial Intelligence

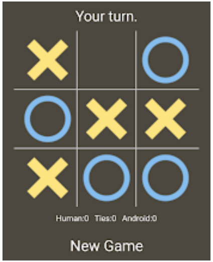
Minimax and Alpha-beta Pruning

Prof. Song Guo, Dr. Desmond Tsoi & Dr. Huiru Xiao

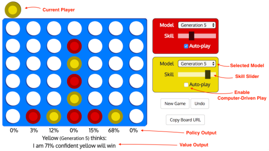
Department of Computer Science & Engineering
The Hong Kong University of Science and Technology, Hong Kong SAR, China

Overview

- In this topic, you will learn how to implement an AI for a 2-player game, Tic-Tac-Toe.
- The AI agent makes use of searching strategies to optimally play the game.
  - Brute-force[†]: Minimax
  - Minimax with Alpha-Beta Pruning
- The same AI strategies can also be applied to other 2-player games, such as Connect-four, Chess, Checkers, Backgammon, etc.



Tic-Tac-Toe



Yellow (Generation 5) thinks:
I am 71% confident yellow will win

Connect-four

Ernest (1676) 23:59

1. e4 c5
2. ♘f3 d6
3. d4 cxd4
4. ♘xd4 ♘f6
5. ♘c3 a6
6. ♗g5

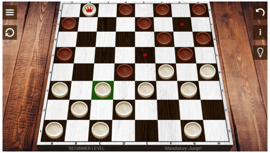
Actions After: Stay

Type /help to show chat commands
Ernest (1676.13) vs richard (1032.42) - 1 day , Rated
Ernest joined
richard joined

Chat:

richard(1032) 23:54

Chess



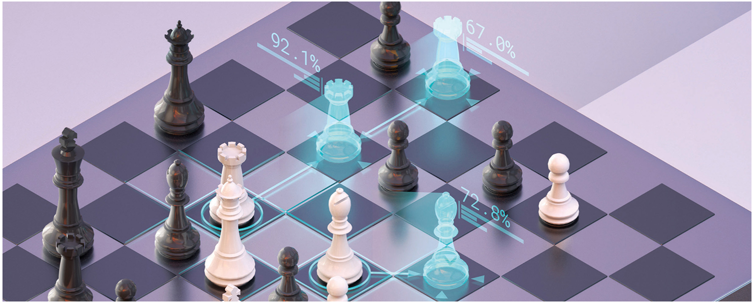
Checkers

Backgammon

[†] Brute force refers to straightforward methods of solving a problem.

Games

- Games are the oldest, most well-studied domain in artificial intelligence.
- Why?
  - They are fun.
  - Easy to represent, rules are clear.
  - Possible combination of move can be big, e.g., there are $\sim 10^{154}$ possible moves in chess.
  - Like the "real world" in that decisions have to be made and time is important.
  - Easy to determine when a program is doing well.



Types of Game

- Perfect vs Imperfect information game
  - Perfect: Player knows all the possible moves of himself and opponent and their results.
  - Imperfect: Player does not know all the possible moves of the opponent.
- Zero-sum vs Non-zero-sum game
  - Zero-sum: A two person game where one player's gain is equal to the other player's loss on any given play of the game.
  - Non-zero-sum: A two person game where player's gain is not necessarily equal to the other player's loss on any given play of the game.
- Deterministic vs Non-deterministic game
  - Deterministic: A game that does not involve random choices.
  - Non-deterministic: A game that involves some random choices.

Examples

	Perfect Information?	Zero-sum?	Deterministic?
Tic-Tac-Toe	Yes	Yes	Yes
Chess	Yes	Yes	Yes
Monopoly	Yes	No	No
Poker	No	Yes	No

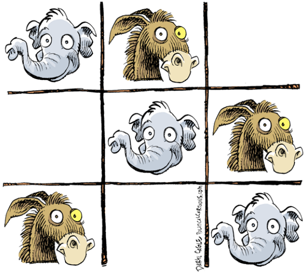
- Tic-Tac-Toe
  - Has perfect information: All players know the moves previously made by all other players.
  - Is zero-sum, as it involves two players, if player 1 wins, she can score 1 and player 2 who losses score (-1) and if the game ends in draw, both players score 0
  - Is deterministic: it is a game that does not involve random choices

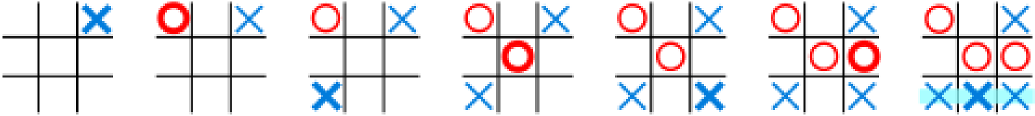
Question

Could you explain the others? For example, why chess game has perfect information, is zero-sum and deterministic?

Tic-Tac-Toe

- Tic-Tac-Toe (also called Noughts and Crosses) is a paper-and-pencil game for two players, who take turns marking the spaces in a 3×3 grid.
- The player who succeeds in placing three of their marks in a diagonal, horizontal, or vertical line is the winner.



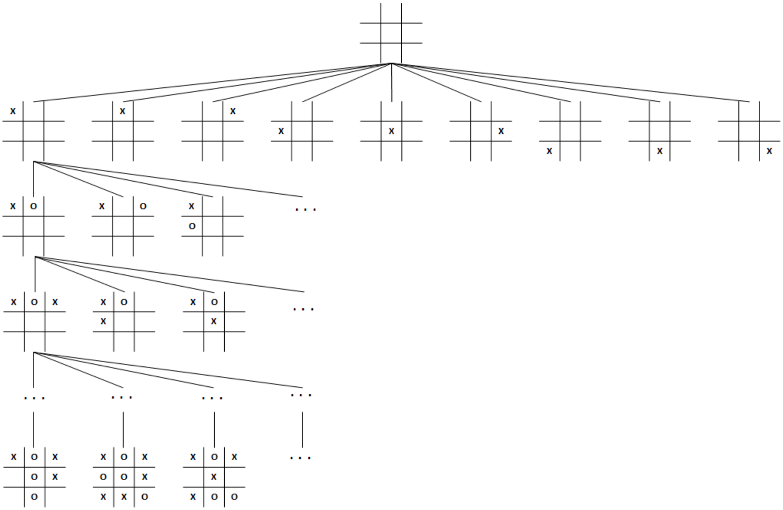


AI - Naïve Approach

- First, if there is a move that will win the game, play it.
- If not, check if there is a move that will block a win from the opposing player. If so, place your marker in the blocking spot.
- Otherwise, just place your marker in a random empty cell.
- The AI may not play optimally, i.e., the AI may not achieve the best results.

AI - Brute Force Approach

Enumerate all the possible states of the game.



- At the very beginning, a player can choose any of the 9 available empty spaces on the game board.
- We can represent these 9 available moves using a Tree as shown on the left.

AI - Brute Force Approach

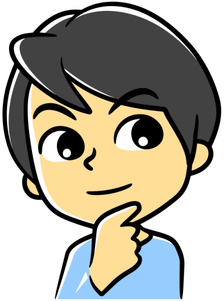
- 1st action: 9 possible choices at the very beginning of the game
- 2nd action: 9×8 (for each 9 possible first choices, we have 8 possible remaining choices)
- 3rd action: 9×8×7 (for each 9×8=72 previous choices we have 7 possible remaining choices)
- ...
- Last action: $9! = 9 \times 8 \times 7 \times \cdots \times 1 = 362880$ choices

$$\begin{aligned}
\textit{Total number of states} &= (9 \times 8 \times 7 \times 6 \times 5 \times 4 \times 3 \times 2 \times 1) + \\
&(9 \times 8 \times 7 \times 6 \times 5 \times 4 \times 3 \times 2) + \\
&(9 \times 8 \times 7 \times 6 \times 5 \times 4 \times 3) + \\
&\dots \\
&(9 \times 8) + \\
&(9) \\
&= 986409
\end{aligned}$$

AI - Brute Force Approach

Questions

1. Is it really to have 986409 possible states in the game?
   Answer: No. The game can end before there is no more empty space on the board.
2. How to determine which move to take based on the current game state?
   Answer: Use minimax. :)



Minimax

- Minimax is a recursive algorithm which is used to choose an optimal move for a player assuming that the other player is also playing optimally.
- In minimax, the two players are called maximizer (denoted as MAX) and minimizer (denoted as MIN), and define a scoring method from the standpoint of the MAX player.
- The maximizer (the AI of Tic-Tac-Toe game) tries to maximize the score while the minimizer (the Human player of Tic-Tac-Toe game) tries to minimize the score of AI.
- It is called minimax because it helps in minimizing that the other players can force us to receive.

Minimax

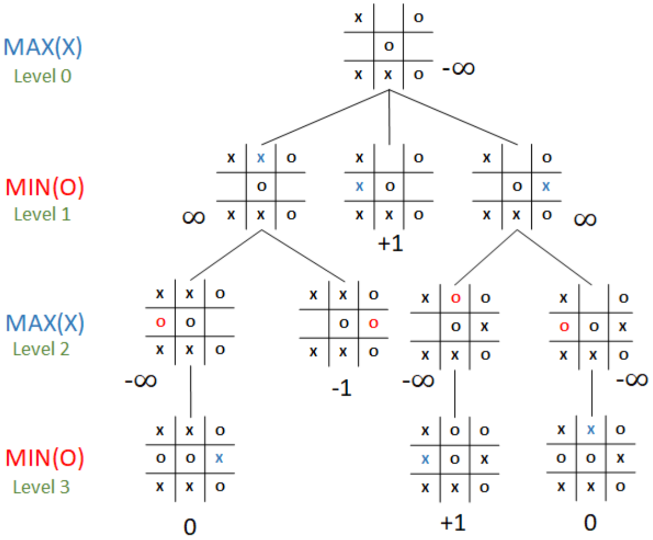
- Steps of the minimax algorithm
  1. Construct the complete game tree
  2. Apply the utility function to each terminal state.
  3. Beginning with the terminal states, determine the utility of the predecessor nodes as follows:
     - Node is a MIN-node: value is the minimum of the children nodes
     - Node is a MAX-node: value is the maximum of the children nodes
  4. From the initial state (i.e., root of the game tree), MAX chooses the move that leads to the highest value (minimax decision)

Search the game tree in a Depth-First-Search manner to find the value of the root. Depth-first search is an algorithm for traversing or searching tree data structures. The algorithm starts at the root node and explores as far as possible along each branch before backtracking.

Utility function

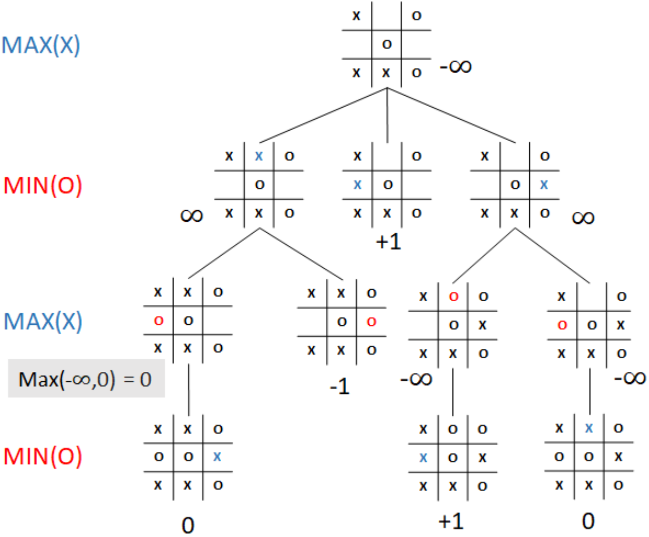
Numeric values given to terminal states.
For example, in Tic-Tac-Toe, win = +1, loss = -1 and draw = 0.

Minimax Example - Initial



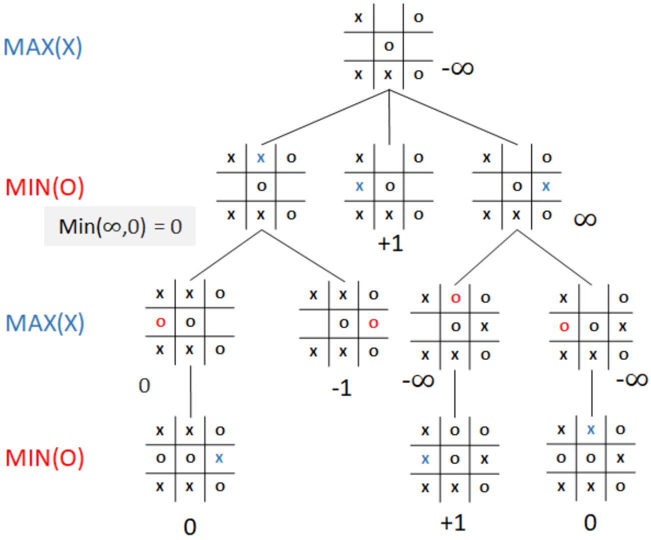
- Terminal states' utility:
  Max wins = 1,
  Max loses = -1,
  Draw = 0
- For maximizier, initialize each node to negative infinity, -∞
- For minimizier, initialize each node to positive infinity, ∞

Minimax Example - Step 1



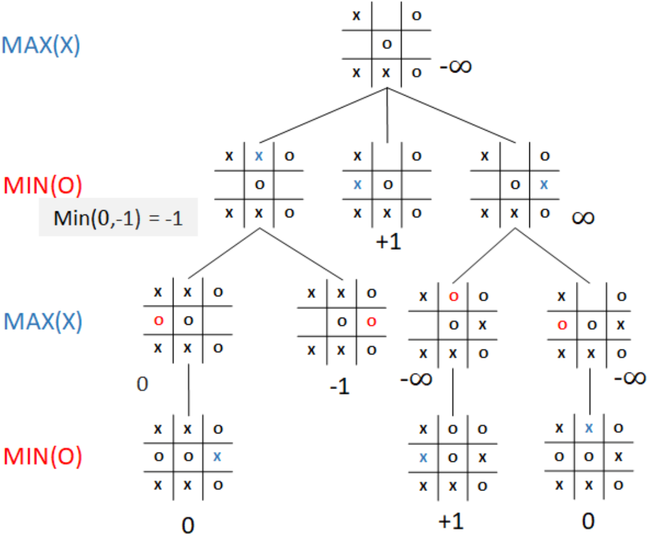
- For the leftmost state in level 2, the maximizer updates its value with the maximum of $-\infty$ and 0, where the value 0 is obtained from its child in level 3.
- So, its value is changed from $-\infty$ to 0.

Minimax Example - Step 2



- For the leftmost state in level 1, the minimizer updates its value with the minimum of ∞ and 0, where the value 0 is obtained from its left child in level 2.
- So, its value is changed from ∞ to 0.

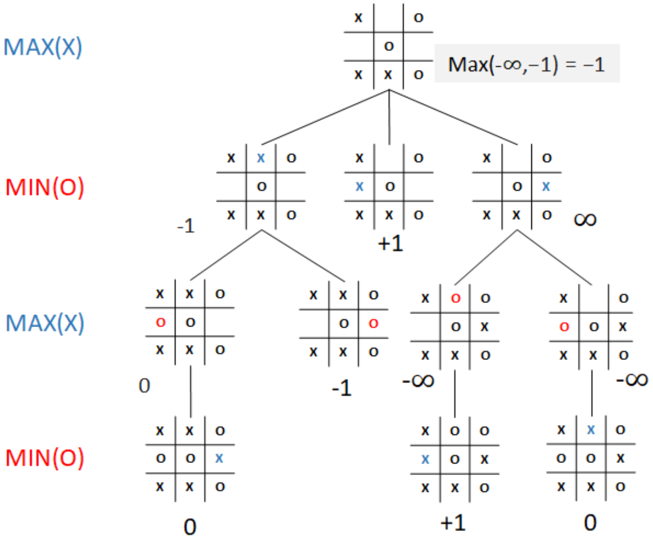
Minimax Example - Step 3



Min(0,-1) = -1

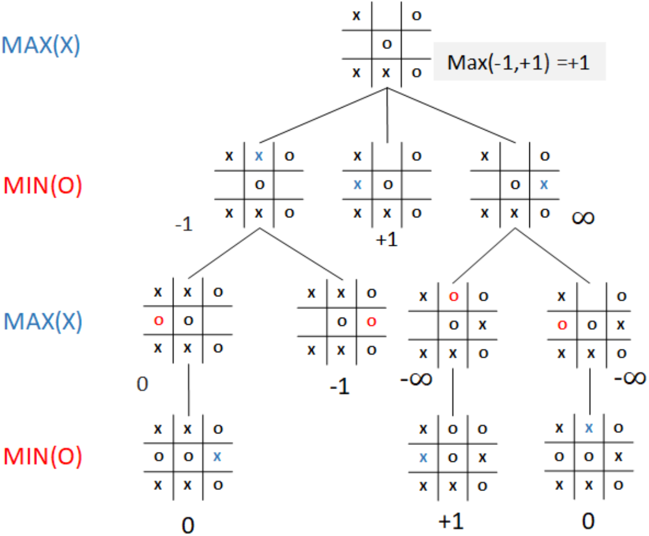
- For the leftmost state in level 1, the minimizer updates its value with the minimum of 0 and -1, where the value -1 is obtained from its right child in level 2.
- So, its value is changed from 0 to -1.

Minimax Example - Step 4



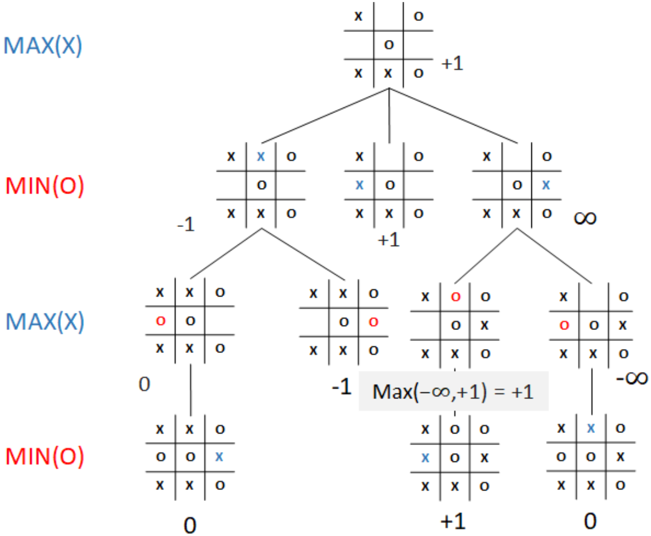
- For the state in level 0, the maximizer updates its value with the maximum of $-\infty$ and -1, where the value -1 is obtained from its left child in level 1.
- So, its value is changed from $-\infty$ to -1.

Minimax Example - Step 5



- For the state in level 0, the maximizer updates its value with the maximum of -1 and $+1$, where the value $+1$ is obtained from its middle child in level 1.
- So, its value is changed from -1 to $+1$.

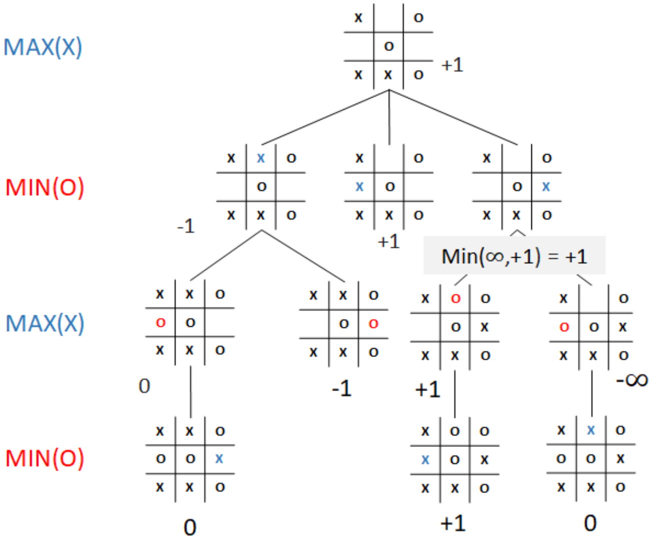
Minimax Example - Step 6



Max(−∞,+1) = +1

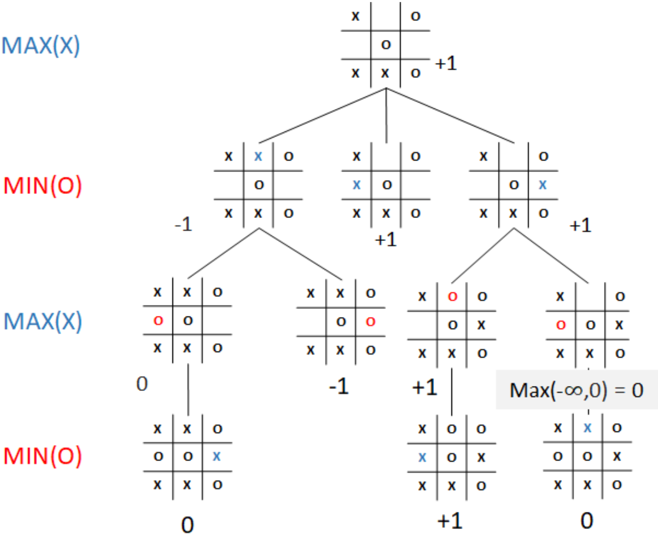
- For the state in level 2, the maximizer updates its value with the maximum of $-\infty$ and $+1$, where the value $+1$ is obtained from its child in level 3.
- So, its value is changed from $-\infty$ to $+1$.

Minimax Example - Step 7



- For the state in level 1, the minimizer updates its value with the minimum of ∞ and $+1$, where the value $+1$ is obtained from its left child in level 2.
- So, its value is changed from ∞ to $+1$.

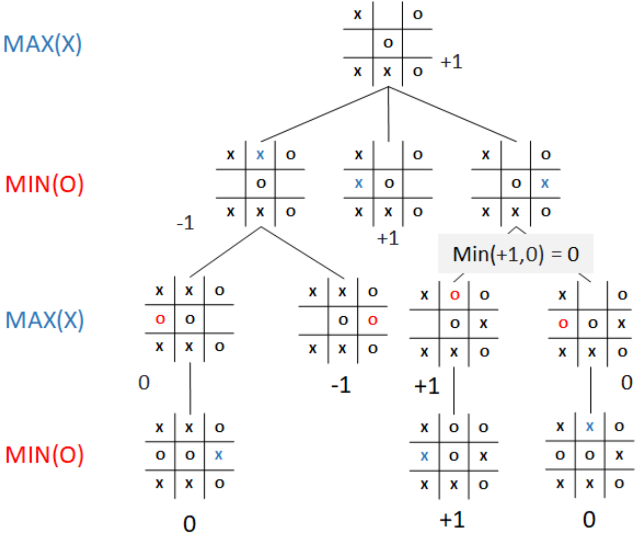
Minimax Example - Step 8



Max(-∞,0) = 0

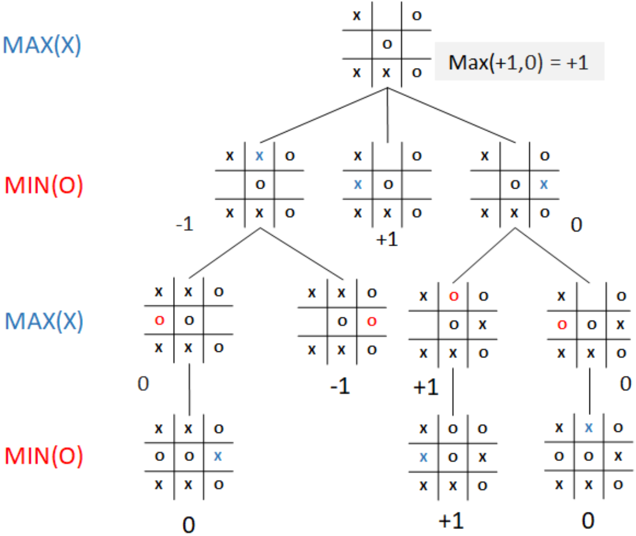
- For the state in level 2, the maximizer updates its value with the maximum of $-\infty$ and 0, where the value 0 is obtained from its child in level 3.
- So, its value is changed from $-\infty$ to 0.

Minimax Example - Step 9



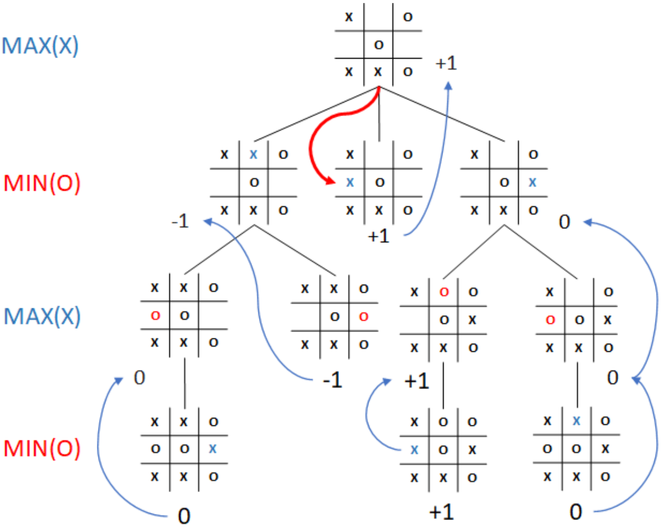
- For the state in level 1, the minimizer updates its value with the minimum of $+1$ and 0, where the value 0 is obtained from its right child in level 2.
- So, its value is changed from $+1$ to 0.

Minimax Example - Step 10



- For the state in level 0, the maximizer updates its value with the maximum of +1 and 0, where the value 0 is obtained from its right child in level 1.
- So, its value remains unchanged, i.e. 0.

Minimax Example - Done



- In the top position, it is X's turn.
- If X plays in the top-center, then O can guarantee a win.
- If X plays in the left center, X will win.
- If X plays right center, then O can force a draw.
- Therefore X will choose to play in the left center.

Minimax Code from Scratch

```
# Draw the current state of the game
def drawboard(board):
    print("Current state: \n\n")
    for i in range (0,9):
        if board[i] == 0:           # If value is 0, it's an empty cell
            print("- ",end = " ")
        if board[i] == 1:           # If value is 1, it's the mark of AI
            print("X ",end = " ")
        if board[i] == -1:          # If value is -1, it's the mark of human
            print("O ",end = " ")
        if (i+1) % 3 == 0:          # Move to the next line after printing 3 elements
            print("\n")
    print("\n")  # Move the next line after printing the game board
```

```
# Check if there is a player wins. If human player wins, return -1.
# If AI player wins, return 1. If draw, return 0. If the game is not finished, return 2.
def check_game(board):
    # There are 8 possible winning patterns.
    # 0, 1, 2 (1st row), 3, 4, 5 (2nd row), 6, 7, 8 (3rd row)
    # 0, 3, 6 (1st col), 1, 4, 7 (2nd col), 2, 5, 8 (3rd col)
    # 0, 4, 8 (Diagonal - top-left to bottom-right)
    # 2, 4, 6 (Diagonal - top-right to bottom-left)
    winning_pos = [ [0,1,2], [3,4,5], [6,7,8], [0,3,6], [1,4,7], [2,5,8], [0,4,8], [2,4,6] ]

    # Check all possible winning patterns for each player.
    for i in range(0,8):
        # Check whether the cell is non-empty AND the winning positions with the same mark
        if board[winning_pos[i][0]] != 0 and\
           board[winning_pos[i][0]] == board[winning_pos[i][1]] and\
           board[winning_pos[i][0]] == board[winning_pos[i][2]]:
           # Fulfiled the winning condition, return the mark of player
           return board[winning_pos[i][2]]

    # When reaching here, it means no player wins yet
    # Check whether there is an empty cell (i.e. with 0)
    # If so, the game is not finished. Return 2
    if any(element == 0 for element in board): return 2
    return 0  # Reach here, it means the game draws
```

```python
# Obtain the user's input and update the board
def human_turn(board):
    pos = input("Enter O's position [1 to 9]: ")
    pos = int(pos)  # Obtain user input, i.e., where to put the mark

    # If the cell picked by the human player is non-empty (i.e., non-zero), illegal
    if board[pos-1] != 0:
       print("Illegal Move!")
       exit(0)      # Terminate the program

    # When reaching here, the move is legal and we put -1 (put the mark of human player)
    board[pos-1] = -1
```

```
# Perform minimax search
def minimax(board, player):
    global count                # Refer to the global variable count in main
    count += 1                  # Make 1 call of minimax already
    result = check_game(board)  # Check if any player wins
    if result != 2:             # If the result is not 2, the game is finished,
       return result            # return 1 if the winner is AI,
                                # otherwise return -1 (i.e., human is the winner)

    scores = []                 # Create an empty list
    for i in range(0,9):        # Check all board locations
        if board[i] == 0:       # If the cell is empty
           board[i] = player    # Try to put the player's mark at cell i+1
           # Perform minimax for the next player, and append its score to the list
           scores.append(minimax(board,player*-1))
           board[i] = 0         # Undo the trial

    # Return the max score if the player is AI or
    # return the min score if the player is human
    return max(scores) if player == 1 else min(scores)
```

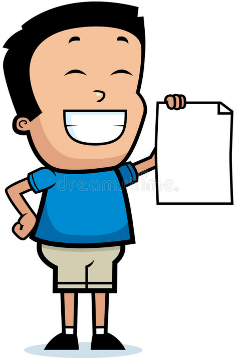
```
# This function makes computer's move based on minimax.
def ai_turn(board):
    pos = -1         # Initialize pos to illegal value, -1, here
    max_value = -2   # Initialize max value so far to -2,
                     # which is a value smaller than the min value possible

    for i in range(0,9):
        if board[i] == 0:              # If the cell is empty
           board[i] = 1                # Try to put X at cell i+1
           score = minimax(board, -1)  # Calculate minimax score for the human player
           board[i] = 0                # Undo the trial
           if score > max_value:       # If we can a better score in the next level,
              max_value = score        # update the score and pos
              pos = i

    board[pos] = 1                     # Put X at pos, which is the best move
```

```
# Main function
# The broad is represented in a single dimensional list
# Initalize the board to all zeros (i.e., all empty cells)
board = [ 0, 0, 0, 0, 0, 0, 0, 0, 0 ]

print("Computer: X VS. You: O")
first = input("Play first (Y/N) :")
if first == 'Y' or first == 'y': # If first is 'Y'/'y', human wants to play first
   play_first = 1
else:
   play_first = 0

for i in range (0,9):            # Play the same. 9 turns in total
    if check_game(board) != 2:   # If the value returned by check_game is not 2,
       break                     # the game is finished

    if (i+play_first) % 2 == 0:  # Take turns to play the game
       count = 0                 # Count the number of minimax calls
       ai_turn(board)
       print("Count:", count)    # Print the count
    else:
       drawboard(board)
       human_turn(board)
```

```
result = check_game(board)       # Check the game again

if result==0:                    # If the result is 0, draw
   drawboard(board)
   print("Draw!!!")
if result==1:                    # If the result is 1, AI wins
   drawboard(board)
   print("AI(X) Wins! Human(O) Loses!")
if result==-1:                   # If the result is -1, human wins
   drawboard(board)
   print("Human(O) Wins! AI(X) Loses!")
```

What if MIN does not play optimally?

- Definition of optimal play for MAX assumes MIN plays optimally, i.e., maximizes worst-case outcome for MAX.
- If MIN does not play optimally, MAX will do even better.



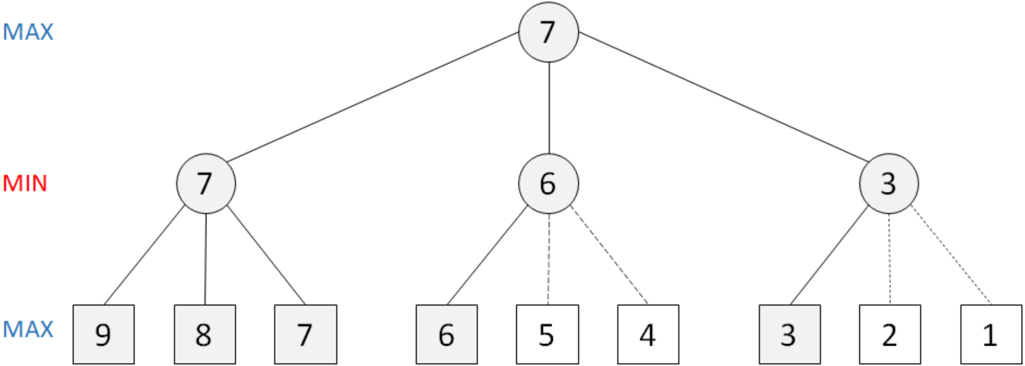
Disadvantages of Minimax

- It has a huge branching factor (the number of children at each node), which makes the process of reaching the goal state slow.
- It can take a long time to evaluate an entire game tree, especially for games such as Go or Chess
- Search and evaluation of unnecessary nodes or branches of the game tree degrades the overall performance and efficiency of the engine.
- Both min and max players have lots of choices to decide from.
- Exploring the entire tree is not possible as there is a restriction of time and space.

Solution: Alpha Beta Pruning

Observations

- The minimax algorithm provides us with the optimal path as expected. But did we really need to do all that work? No!
- Let's demonstrate it using an example.



Explanations

- Let's pretend that we are at the middle level stage of finding the minimum value between 6, 5, and 4.
- We have already calculated the value of node 6. So, we are AT LEAST guaranteed that the minimum value between these 3 nodes is $<=$ 6.
- On the level above (top level), we are trying to maximize the values. We've already calculated a value of 7 to the left, something that is guaranteed to be 6 or less will NEVER BE GREATER than 7.
- So, we don't need to go any further with the calculations.
- How about the minimum value between 3, 2, 1? Same!

Alpha-Beta Pruning

Idea

Prevent ourselves from exploring branches of the game tree that are not worth exploring (because they will have no effect on the final outcome).

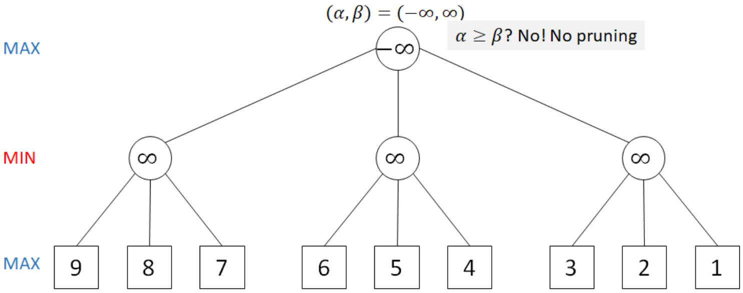
Alpha and Beta

- We define two values α and β to store information about the potential of a branch for us or the opponent.
  - α-cutoff: Terminate branch because AI already has a better opportunity elsewhere.
  - β-cutoff: Terminate branch because opponent (human) already has better opportunity elsewhere.
- α is the best (highest) found so far along the path for Max.
- β is the best (lowest) found so far along the path for Min.
- We can prune a branch if $\alpha \geq \beta$.

Procedure of Alpha-Beta Pruning

- We define two values α and β to store information about the potential of a branch for us or the opponent.
- The variable α and β are used to keep track of the current upper and lower bounds.
- Pass current values of α and β of parent down to child nodes during search.
- Update values of α and β during search:
  - For MAX, compare α with the current node value (v). If $\alpha < v$, $\alpha = v$, otherwise α remains unchanged.
  - For MIN compare β with the current node value (v). If $\beta > v$, $\beta = v$, otherwise β remains unchanged.
- Prune remaining branches at a node when $\alpha \geq \beta$.
- Update values of α and β of parent using the value returned by the child.
  - If parent node is MAX, compare α with the value of child (v). If $\alpha < v$, $\alpha = v$, otherwise α remain unchanged.
  - If parent node is MIN, compare β with the value of child (v). If $\beta > v$, $\beta = v$, otherwise β remain unchanged.

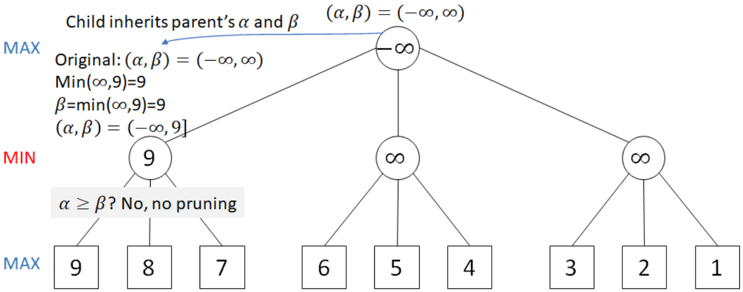
Alpha-Beta Pruning Example



$(\alpha, \beta) = (-\infty, \infty)$
$\alpha \geq \beta$? No! No pruning
MAX
MIN
MAX
$-\infty$
∞
∞
∞
9
8
7
6
5
4
3
2
1

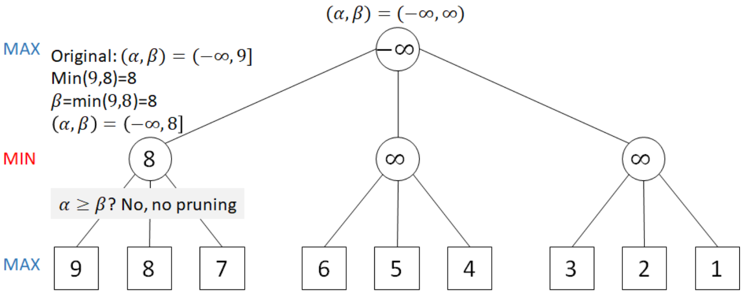
- At the first step, the Max player will initialize α to $-\infty$, and β to ∞.
- We check whether $\alpha \geq \beta$. It is not and therefore no pruning is done.

Alpha-Beta Pruning Example


Child inherits parent's α and β
$(\alpha,\beta)=(-\infty,\infty)$
MAX
$-\infty$
Original: $(\alpha,\beta)=(-\infty,\infty)$
Min(∞,9)=9
β=min(∞,9)=9
$(\alpha,\beta)=(-\infty,9]$
MIN
9
∞
∞
$\alpha\geq\beta$? No, no pruning
MAX
9
8
7
6
5
4
3
2
1

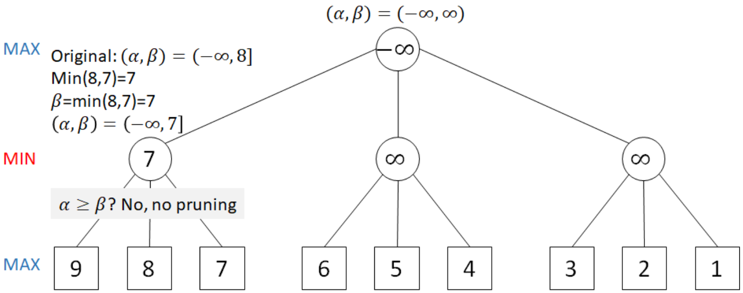
- The Max player moves from root node to its left child by passing down $\alpha = -\infty$ and $\beta = \infty$ to it.
- At the left node of the middle level, we do the following with the first terminal value 9.
  - Node value = Min(node value, 9), i.e. Min(∞,9) = 9.
  - β = Min(β, 9), i.e. Min(∞, 9) = 9.
- We check whether $\alpha \geq \beta$. It is not and therefore no pruning is done.

Alpha-Beta Pruning Example


$(\alpha, \beta) = (-\infty, \infty)$
MAX
Original: $(\alpha, \beta) = (-\infty, 9]$
Min(9,8)=8
β=min(9,8)=8
$(\alpha, \beta) = (-\infty, 8]$
$-\infty$
MIN
8
∞
∞
$\alpha \geq \beta$? No, no pruning
MAX
9
8
7
6
5
4
3
2
1

- The Max player moves from root node to its middle child.
- At the left node of the middle level, we do the following with the second terminal value 8.
  - Node value = Min(node value, 8), i.e. Min(9,8) = 8.
  - β = Min(β, 8), i.e. Min(9, 8) = 8.
- We check whether $\alpha \geq \beta$. It is not and therefore no pruning is done.

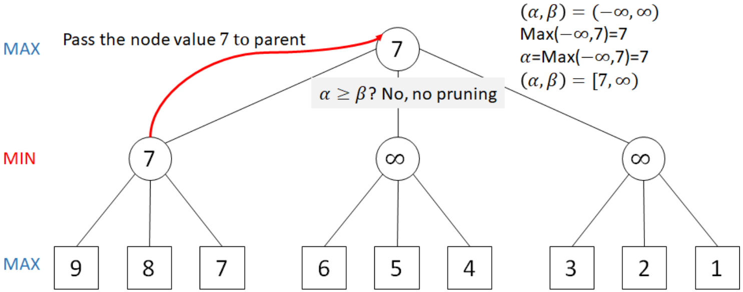
Alpha-Beta Pruning Example



$(\alpha,\beta) = (-\infty,\infty)$
MAX
Original: $(\alpha,\beta) = (-\infty,8]$
Min(8,7)=7
β=min(8,7)=7
$(\alpha,\beta) = (-\infty,7]$
$-\infty$
MIN
7
∞
∞
$\alpha \geq \beta$? No, no pruning
MAX
9
8
7
6
5
4
3
2
1

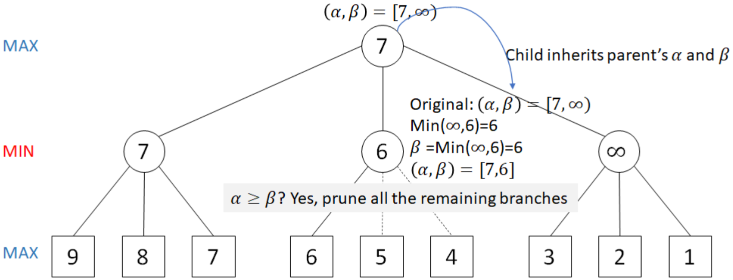
- The Max player moves from root node to its right child.
- At the left node of the middle level, we do the following with the second terminal value 7.
  - Node value = Min(node value, 7), i.e. Min(8,7) = 7.
  - β = Min(β, 7), i.e. Min(8, 7) = 7.
- We check whether $\alpha \geq \beta$. It is not and therefore no pruning is done.

Alpha-Beta Pruning Example


MAX
Pass the node value 7 to parent
7
$(\alpha, \beta) = (-\infty, \infty)$
Max($-\infty$,7)=7
α=Max($-\infty$,7)=7
$(\alpha, \beta) = [7, \infty)$
$\alpha \geq \beta$? No, no pruning
MIN
7
∞
∞
MAX
9
8
7
6
5
4
3
2
1

- Now, the algorithm backtrack to the root, where the value of the middle left node, i.e. 7, is passed to the parent.
- At the root node, we do the following with the returned value 7.
  - Node value = Max(node value, 7), i.e. Max($-\infty$, 7) = 7
  - α = Max(α, 7), i.e. Max($-\infty$, 7) = 7
- We check whether $\alpha \geq \beta$. It is not and therefore no pruning is done.

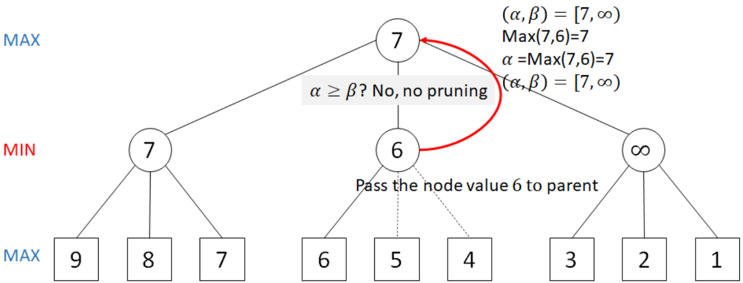
Alpha-Beta Pruning Example



$(\alpha, \beta) = [7, \infty)$
Child inherits parent's α and β
MAX
MIN
MAX
7
7
6
∞
Original: $(\alpha, \beta) = [7, \infty)$
Min(∞,6)=6
β =Min(∞,6)=6
$(\alpha, \beta) = [7,6]$
$\alpha \geq \beta$? Yes, prune all the remaining branches
9
8
7
6
5
4
3
2
1

- The Max player moves from root node to its middle child by passing down $\alpha = 7$ and $\beta = \infty$ to it.
- At the middle node of the middle level, we do the following with the first terminal value 6.
    - Node value = Min(node value, 6), i.e. Min(∞,6) = 6.
    - β = Min(β, 6), i.e. Min(∞, 6) = 6.
- We check whether $\alpha \geq \beta$. We notice that $\alpha = 7$ is greater than $\beta = 6$, and therefore pruning is done.

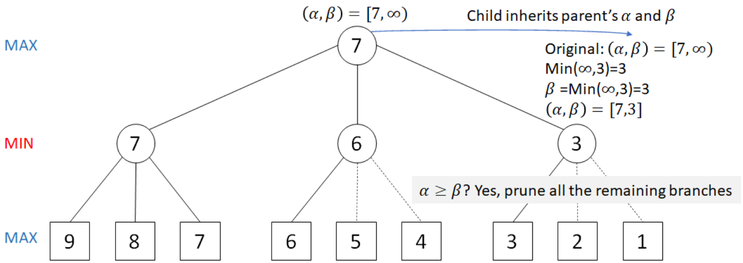
Alpha-Beta Pruning Example



$(\alpha, \beta) = [7, \infty)$
Max(7,6)=7
α =Max(7,6)=7
$(\alpha, \beta) = [7, \infty)$
MAX
7
$\alpha \geq \beta$? No, no pruning
MIN
7
6
∞
Pass the node value 6 to parent
MAX
9
8
7
6
5
4
3
2
1

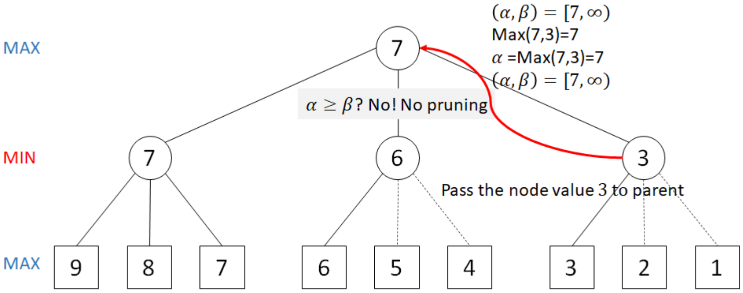
- Now, the algorithm backtrack to the root, where the value of the center node of the middle layer, i.e. 6, is passed to the parent.
- At the root node, we do the following with the returned value 6.
    - Node value = Max(node value, 6), i.e. Max(7, 6) = 7
    - α = Max(α, 7), i.e. Max(7, 6) = 7
- We check whether $\alpha \geq \beta$. It is not and therefore no pruning is done.

Alpha-Beta Pruning Example


$(\alpha, \beta) = [7, \infty)$
Child inherits parent's α and β
MAX
MIN
MAX
7
7
6
3
9
8
7
6
5
4
3
2
1
Original: $(\alpha, \beta) = [7, \infty)$
Min(∞,3)=3
β =Min(∞,3)=3
$(\alpha, \beta) = [7,3]$
$\alpha \geq \beta$? Yes, prune all the remaining branches

- The Max player moves from root node to its right child by passing down $\alpha = 7$ and $\beta = \infty$ to it.
- At the right node of the middle level, we do the following with the first terminal value 3.
  - Node value = Min(node value, 3), i.e. Min(∞,3) = 3.
  - β = Min(β, 3), i.e. Min(∞, 3) = 3.
- We check whether $\alpha \geq \beta$. We notice that $\alpha = 7$ is greater than $\beta = 3$, and therefore pruning is done.

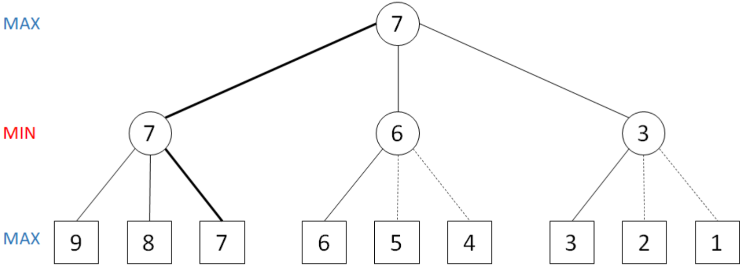
Alpha-Beta Pruning Example



$(\alpha, \beta) = [7, \infty)$
Max(7,3)=7
α =Max(7,3)=7
$(\alpha, \beta) = [7, \infty)$
$\alpha \geq \beta$? No! No pruning
Pass the node value 3 to parent
MAX
MIN
MAX
7
7
6
3
9
8
7
6
5
4
3
2
1

- Now, the algorithm backtrack to the root, where the value of the right node of the middle layer, i.e. 3, is passed to the parent.
- At the root node, we do the following with the returned value 3.
    - Node value = Max(node value, 3), i.e. Max(7, 3) = 7
    - α = Max(α, 3), i.e. Max(7, 3) = 7
- We check whether $\alpha \geq \beta$. It is not and therefore no pruning is done.

Alpha-Beta Pruning Example



- The best value for the root node is 7. The figure on the left shows the nodes which are computed and the nodes which has never computed. Hence the optimal value for the maximizer is 7 for this example.

Alpha-Beta Pruning Code

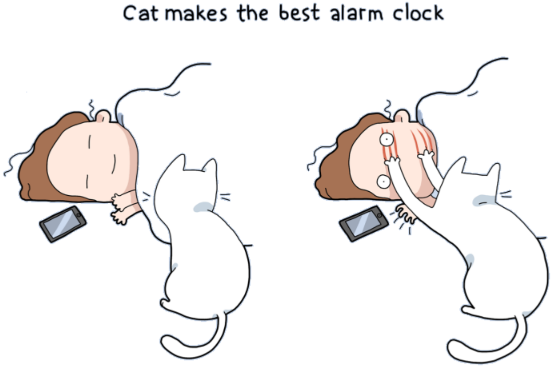
Replace the minimax function that we defined above with the following:

```
# Perform minimax search with alpha-beta pruning
def minimax_abp(board, player, alpha = -float('inf'), beta = float('inf')):
    global count                # Refer to the global variable count in main
    count += 1                  # Make 1 call of minimax_abp already
    result = check_game(board)  # Check if any player wins
    if result != 2:             # If the result is not 2, the game is finished,
       return result            # return 1 if the winner is AI,
                                # otherwise return -1 (i.e., human is the winner)
    if player == 1:
       value = -float('inf')
       for i in range(0,9):                  # Check all board locations
           if board[i] == 0:                 # If the cell is empty
              board[i] = player              # Try to put the player's mark at cell i+1
              # Perform minimax with alpha-beta pruning for the next player
              # and update current value if needed
              value = max(value, minimax_abp(board, player*-1, alpha, beta))
              alpha = max(alpha, value)  # Update alpha with max of current alpha and value
              board[i] = 0                   # Undo the trial
              if alpha >= beta: break
```

```
else:
   value = float('inf')
   for i in range(0,9):              # Check all board locations
       if board[i] == 0:             # If the cell is empty
          board[i] = player          # Try to put the player's mark at cell i+1
          # Perform minimax with alpha-beta pruning for the next player
          # and update current value if needed
          value = min(value, minimax_abp(board, player*-1, alpha, beta))
          beta = min(beta, value)    # Update beta with min of current beta and value
          board[i] = 0               # Undo the trial
          if alpha >= beta: break

# Return the max score if the player is AI or
# return the min score if the player is human
return value
```

Advantages of Alpha-Beta Pruning

- It plays a great role in reducing the number of nodes which are found out by minimax.
- When one chance or option is found at the minimum, it stops assessing a move.
- This method also helps to improve the search procedure in an effective way.



Disadvantages of Alpha-Beta Pruning

- Alpha-beta prunes large part of search space, but still needs to search all the way to terminal states.
- Note that for some games, the moves must be made in reasonable amount of time.

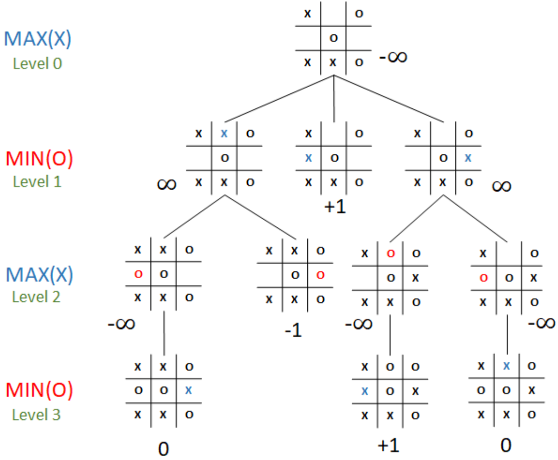
Approaches to Remedy the Problem

- Set a depth limit
- Heuristic evaluation function: Estimated desirability or utility of position
  $\Rightarrow$ Correlate with the actual chance of winning.

Alpha-Beta Pruning

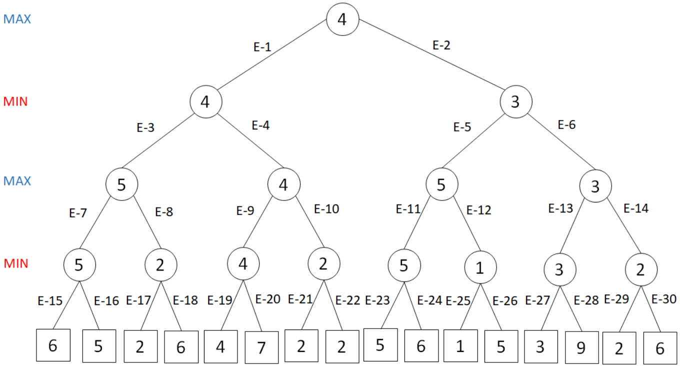
Question

Can you find the optimal move for the Tic-Tac-Toe example using Alpha-Beta Pruning? :)



Practice Problem

Consider the game tree below filled with the values returned by the standard minimax algorithm.



(a) State the best initial move for the first player (i.e. E-1 or E-2).

(b) Apply alpha-beta pruning on the given tree and state the branches that need not be examined or considered by putting down the edge label(s) (e.g. E-1, E-2, E-3, . . ., E-30).

Practice Problem

(a) E-1

(b) E-18, E-22, E-26, E-28 and E-30.

That's all!

Any questions?