# Introduction

- **Connect 4 as an AI problem:**

Connect 4 is a zero-sum or deterministic game played on a vertical grid of six rows and seven columns. Each player has an equal number of chips (21) initially to drop one at a time from the top of the board. they will take turns playing and whoever makes a line of four chips either vertically, horizontally, or diagonally wins.

**Problem formulation:**

1- **Initial state**: empty grid with six rows and seven columns

2- **Players**: person/random function and agent(min-max OR alpha-beta pruning)

3- **Actions**: drop a chip from the top of the board in an empty slot

4- **Result**: chip is inserted into a slot

5- **Terminal test**: a player has four chips in a row either vertically, horizontally, or diagonally

6- **Utility function**: -10000, 10000, 0

- **Algorithms used to tackle this problem:**

**1- Min-Max algorithm:**

Since this game is categorized as a zero-sum game, the minimax algorithm, which is a decision rule used in AI, can be applied. The Minimax search algorithm takes the form of a search tree, where each level of the tree represents each player's respective turn. Two players, Max and Min, favor high and low heuristic values respectively

- **Pseudocode:**

```
class minmax extends connect4AI{
    depth
    X<--0
    Minmax(int depth){
        this.depth=depth
}
    getAction(State st){
        Val<-- max(st, depth)
        Return x
```

```
}
    max(State st, int d){
            children <-- new ArrayList<Integer>()
            IF(d ==0)
                return st.evaluationFunction()
            ELSE
                children <-- st.getLegalActions()
                heuristic <-- -10000000

               changingheuristic
               z
                FOR i <-- 0 to children.size() do

                  changingheuristic <-- min(st.generateSuccessor('O',children.get(i)),d)
                  IF changingheuristic >= heuristic

                      heuristic <--changingheuristic
                     this.x <-- i
                  END IF


                  return heuristic
}

 min(State st, int d) {
              children <-- new ArrayList<Integer>()
             IF d == 0
               return st.evaluationFunction()
             ELSE IF

               children <-- st.getLegalActions()
```

```
        heuristic <-- 10000000
        x <-- 0
       changingheuristic
       FOR  i <--0  to children.size() do

          changingheuristic <-- max(st.generateSuccessor('X',children.get(i)),d-1)
          IF changingheuristic <= heuristic
              heuristic <-- changingheuristic
          END IF

         return heuristic
}
}
```

### 2- Alpha-Beta pruning algorithm:

Another algorithm used to solve this problem is an optimization of minimax known as alpha-beta pruning. Alpha Beta pruning works by reducing the number of nodes that need to be explored in the minimax tree. This is accomplished by working with the idea that player Min will try to pick the lowest value, and Max will try to pick the highest value.

- **Pseudocode:**

```
class alphabetapruning extends connect4AI{
    gameover(State s, int depth){
        return (s.isGoal('X') | s.isGoal('O') | depth == 0)
  }

    minimaxAlgoPruning(State s, int depth, int alpha, int beta, boolean AI){
        legallocations <-- s.getLegalActions()
        size <-- legallocations.size()
       Changingheuristic <-- 0
       heuristic <-- 0
       Best_Position <-- 0
        player <-- ' '
        IF AI
```

```
        player <-- 'O'
    ELSE
        player <-- 'X'
    IF gameover(s,depth)
        IF s.isGoal('X')
            return 10000
        ELSE IF s.isGoal('O')
            return -10000
        ELSE
            return s.evaluationFunction()
    IF AI
        heuristic <-- Integer.MIN_VALUE
        ran <-- new Random()
        rand <-- ran.nextInt(size)
        Best_Position <-- legallocations.get(rand)
        For c <-- 0 to size do
            Changingheuristic <-- (int) minimaxAlgoPruning((s.generateSuccessor(player,
legallocations.get(c))),(int)depth-1, (int)alpha, (int)beta, false)
            alpha <-- (int) Math.max(alpha, heuristic)
            IF alpha >= beta
                break;
            END IF
            IF Changingheuristic > heuristic
                heuristic <-- Changingheuristic
                Best_Position <-- c
            END IF
        return Best_Position

ELSE

    heuristic <-- Integer.MAX_VALUE
    ran <-- new Random()
```

```
                rand <-- ran.nextInt(size)
                Best_Position <-- legallocations.get(rand)
                for c <-- 0 to size do
                    Changingheuristic <-- (int) minimaxAlgoPruning(s.generateSuccessor(player,
legallocations.get(c)),(int)depth-1,(int)alpha,(int)beta, true)
                    beta <-- (int) Math.min(beta, heuristic)
                    IF  alpha >= beta
                        break
                    END IF

                    IF Changingheuristic < heuristic
                        heuristic <-- Changingheuristic
                        Best_Position <-- c
                    END IF
                return Best_Position

    return 0
}
```

# Implementation

This code was implemented using **Java** and contains four classes:

- **Class State:**

<u>This class includes the following methods:</u>

1. **public State(int n_rows, int n_cols):** Basic method for constructing the game by creating a two demensional array.
2. **public boolean equals(Object obj):** method used for checking if two boards are the same.
3. **public int hashCode():** method that returns the hashCode.
4. **public Object clone():** method that copies the current board into a new state.

5. **public ArrayList getLegalActions():** Returns a list of actions that can be taken from the current state. Actions are integers representing the column where a coin can be dropped.
6. **public State generateSuccessor(char agent, int action):** Returns a State object that is obtained by the agent and performs an action on the current state.
7. **public void printBoard():** Prints the current state of the game.
8. **public boolean isGoal(char agent):** Returns True/False if the agent has won the game by checking all rows/columns/diagonals for a sequence of >=4.
9. **public double evaluationFunction():** returns the value of each state for minimax to min/max over at zero depth. If the goal is 'O' it will return 1000, if the goal is 'X' it will return –1000, otherwise it will return 0.

- **Class connect4AI:**

<u>This class contains our main method which does the following:</u>

- Asks the user to insert the depth.
- Creates two objects, one of type minmax, the other of type alphabetapruning.
- Creates 4 states, each with 6 rows and 7 columns.
- A switch with the following options:

1. User vs minmax :

When the user chooses 1, the program will have the user play against minmax and it will calculate the start time and end time.

2. Random vs minmax :

When the user chooses 2, the program will have minmax play against random moves and it will calculate the start time and end time.

3. User vs alpha-beta :

When the user chooses 3, the program will have the user play against alphabetapruning and it will calculate the start time and end time.

4. Random vs alpha-beta:

When the user chooses 4, the program will have alphabetapruning play against random moves and it will calculate the start time and end time.

5. The time:

When the user chooses 5, the program will print total time(end time – start time) for each game.

6. Exit the program.

When the user chooses 6, the program will print a message and terminate.

- **Class minmax:**

This class is our minmax algorithm and includes the following methods:

- **public minmax(int depth):** minmax constructor
- **public int getAction(State st) throws CloneNotSupportedException:** returns the max value at the inserted depth by calling the max method, which is the action of the agent
- **public double max(State st, int d) throws CloneNotSupportedException:** the program will check which actions are legal, use the evaluation function if the game has ended, and return the best choice of position for the maximizer
- **public double min(State st, int d) throws CloneNotSupportedException:** the program will check which actions are legal, use the evaluation function if the game has ended, and return the best choice of position for the minimizer

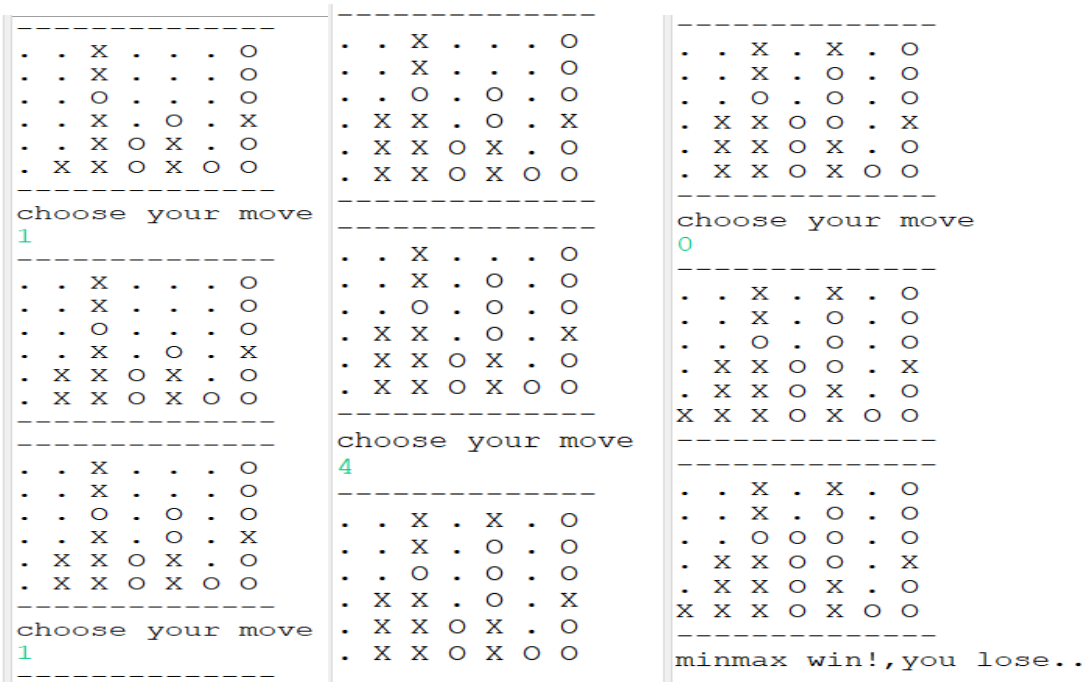
- **Class alphabetapruning:**
- **boolean gameover(State s, int depth):** checks if the game has ended.
- **double minimaxAlgoPruning(State s, int depth, int alpha, int beta, boolean AI):** the program will return alpha-beta's decision by checking which actions are legal, using the evaluation function if the game has ended, and returning the best position by comparing heuristics based on whether it is minimizing or maximizing. It will also stop searching through sibling nodes whenever the changing heuristic is worse than the previous heuristic.

# Result & discussion

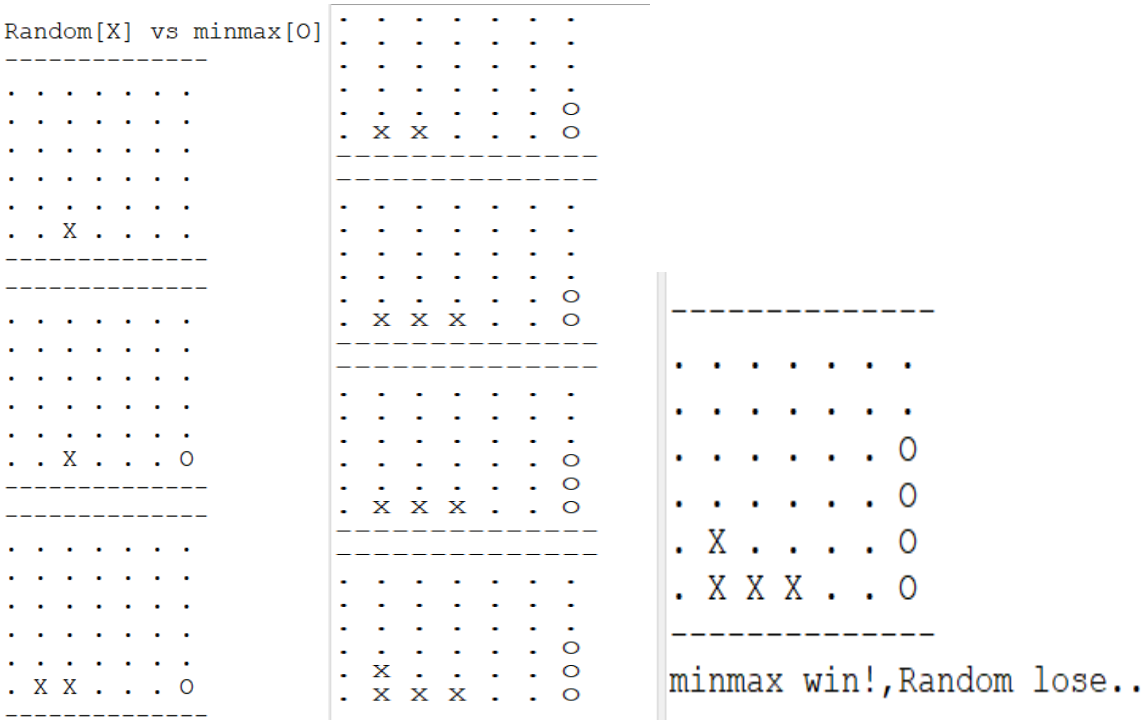
- **RUN:** depth = 2

### 1- User vs minmax:

```
User[X] vs minmax[O]
-------------
. . . . . . .
. . . . . . .
. . . . . . .
. . . . . . .
. . . . . . .
. . . . . . O
-------------
choose your move
4
-------------
. . . . . . .
. . . . . . .
. . . . . . .
. . . . . . .
. . . . . . .
. . . . X . O
-------------
-------------
. . . . . . .
. . . . . . .
. . . . . . .
. . . . . . .
. . . . . . O
. . . . X . O
-------------
choose your move
6
-------------
. . . . . . .
. . . . . . O
. . . . . . O
. . . . . . X
. . X . X . O
. . X O X . O
-------------
-------------
. . . . . . O
. . . . . . O
. . . . . . O
. . . . . . X
. . X . X . O
. . X O X . O
-------------
choose your move
1
-------------
. . . . . . O
. . . . . . O
. . . . . . O
. . . . . . X
. . X . X . O
. X X O X . O
-------------
-------------
. . . . . . .
. . . . . . .
. . . . . . .
. . . . . . X
. . . . . . O
. . . . X . O
-------------
-------------
. . . . . . .
. . . . . . .
. . . . . . O
. . . . . . X
. . . . X . O
. . . . X . O
-------------
choose your move
4
-------------
. . . . . . .
. . . . . . .
. . . . . . O
. . . . . . X
. . . . X . O
. . . . X . O
-------------
-------------
. . . . . . O
. . . . . . O
. . . . . . O
. . . . . . X
. . X O X . O
. X X O X . O
-------------
choose your move
2
-------------
. . . . . . O
. . . . . . O
. . . . . . O
. . X . . . X
. . X O X . O
. X X O X . O
-------------
-------------
. . . . . . O
. . X . . . O
. . O . . . O
. . X . . . X
. . X O X . O
. X X O X . O
-------------
choose your move
2
-------------
. . . . . . O
. . . . . . O
. . . . . . O
. . X . . . X
. . X O X . O
. X X O X . O
-------------
-------------
. . . . . . O
. . X . . . O
. . O . . . O
. . X . . . X
. . X O X . O
. X X O X O O
-------------
choose your move
2
-------------
. . X . . . O
. . X . . . O
. . O . . . O
. . X . . . X
. . X O X . O
. X X O X O O
```

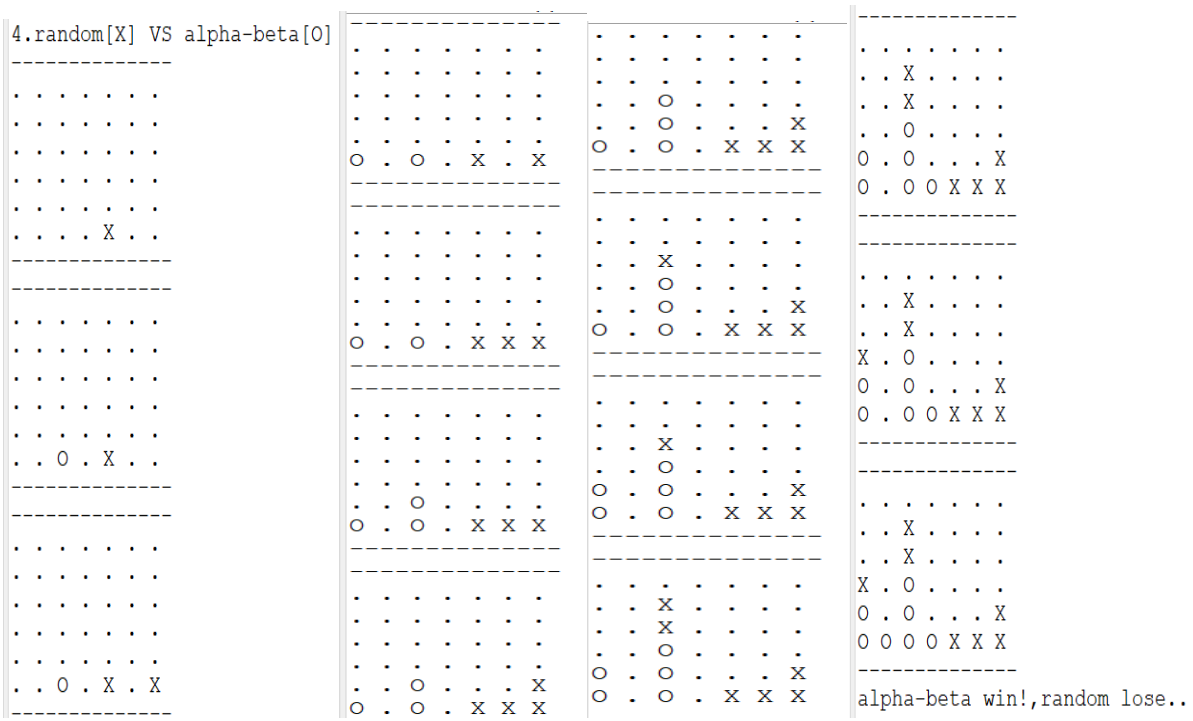



## 2- Random vs minmax:



### 3- User vs alpha-beta:

### 4- Random vs alpha-beta:




### 5- Time:

```
Time in milli seconds: (User vs minmax ) = 44631
Time in milli seconds: (Random vs minmax) = 3554
Time in milli seconds: (User vs alpha-beta) = 21024
Time in milli seconds: (Random vs alpha-beta) = 10
```

- **Discussion:**

We have observed that the alpha-beta algorithm is more efficient than the minimax algorithm since, as shown in the previous results, minimax took 44631 milli seconds to complete a game against the user while alpha-beta only took 21024 milliseconds. Also, when playing against random, alpha-beta was able to finish in just 10 milliseconds, as opposed to minimax which needed 3554 milliseconds. Therefore, alpha-beta will always take less time to complete a game, making it the more optimal algorithm. However, the best algorithm for tackling this problem would be MTD(n,f) which stands for Memory-enhanced Test Driver with node ‘n’ and value ‘f’. MTD(f) is an alpha-beta game tree search algorithm modified to use ‘zero-window’ initial search bounds, and memory (usually a transposition table) to reuse intermediate search results. MTD(f) on average is faster and evaluates fewer leaf nodes than alpha-beta, especially at a deeper depth.

# Conclusion

The Connect4 problem can be solved using several different game playing algorithms, the most widely used being minimax and its optimization, alpha-beta pruning, with alpha-beta pruning being the more efficient algorithm for its reduction of time. We have faced quite a few difficulties when implementing these algorithms, including comparing each algorithms total time when playing against a user, and pruning the search tree in the alpha-beta algorithm. Also, through research we have found that MTD(f) would be the most optimal algorithm used in developing an agent that plays the connect 4 game.