PROBLEM STATEMENT :

Implementation of MiniMax approach for TIC-TAC-TOE using Java/ Scala/ Python-Eclipse Use GUI. Player X and Player O are using their mobiles for the play. Refresh the screen after the move for both the players.

OBJECTIVE :

• To apply algorithmic strategies while solving problems
• To develop time and space efficient algorithms
• To study algorithmic examples in distributed, concurrent and parallel environments

THEORY :

Given a game tree, the optimal strategy can be determined from the minimax value of each node, which we write as MiNimAx(n). The minimax value of a node is the utility (for MAX) of being in the corresponding state, assuming that both players play optimally from there to the end of the game. Obviously, the minimax value of a terminal state is just its utility.

Furthermore, given a choice, MAX prefers to move to a state of maximum value, whereas MIN prefers a state of minimum value. So we have the following:

MINIMAX(s) =

{	**UTILITY (S)**	**if TERMINAL-TEST(s) }**
{	**$\max_a \varepsilon$ Actions(s) MINIMAX(RESULT(s,a))**	**if PLAYER(S)-----MAX }**
{	**$\max_a \varepsilon$ Actions(s) MINIMAX(RESULT(s,a))**	**if PLAYER(S)-----MAX }**

The minimax algorithm computes the minimax decision from the current state. It uses a simple recursive computation of the minimax values of each successor state, directly implementing the defining equations. The recursion proceeds all the way down to the leaves of the tree, and then the minimax values are backed up through the tree as the recursion unwinds.

For two player games, the minimax algorithm is such a tactic, which uses the fact that the two players are working towards opposite goals to make predictions about which future states will be reached as the game progresses, and then proceeds accordingly to optimize its chance of victory. The theory behind minimax is that the algorithm's opponent will be trying to minimize whatever value the algorithm is trying to maximize (hence, "minimax"). Thus, the computer should make the move which leaves its opponent capable of doing the least damage.

In the ideal case (i.e., the computer has infinite time and infinite storage capacity), the computer will investigate every game outcome possible from the game's current state (as well as the paths taken to reach those states) and assign those outcomes values. For a win-or-lose game like chess or tic-tac-toe, there are only three possible values-win, lose, or draw (often assigned numeric values of 1, -1, and 0 respectively)--but for other games like backgammon or poker with a "score" to maximize, the scores themselves are used. Then, starting from the bottom, the computer evaluates which possible outcome is best for the computer's opponent. It then assumes that, if that game stage is reached, its opponent will make the move which leads to the outcome best for the opponent (and

worse for the computer). Thus, it can "know" what its opponent will do, and have a concrete idea of what the final game state will be if that second-to-last position is in fact reached. Then, the computer can treat that position of a terminal node of that value, even though it is not actually a terminal value. This process can then be repeated a level higher, and so on. Ultimately, each option that the computer currently has available can be a assigned a value, as if it were a terminal state, and the computer simply picks the highest value and takes that action.

Describing a Perfect Game of Tic Tac Toe

To begin, let's start by defining what it means to play a perfect game of tic tac toe:

If I play perfectly, every time I play I will either win the game, or I will draw the game. Furthermore if I play against another perfect player, I will always draw the game.

How might we describe these situations quantitatively? Let's assign a score to the "end game conditions:"

- Player wins.
- Player loses.
- The game is a draw.

A description for the algorithm, assuming X is the "turn taking player," would look something like:

- If the game is over, return the score from X's perspective.
- Otherwise get a list of new game states for every possible move
- Create a scores list
- For each of these states add the minimax result of that state to the scores list
- If it's X's turn, return the maximum score from the scores list
- If it's O's turn, return the minimum score from the scores list

ALGORITHM :

```
function MINIMAX (N) :
begin
if N is deep enough then
        return the estimated score of this leaf
else
        Let N1,N2,..... Nm  be the successors of N;
        if N is a Min node then :
                return min{MINIMAX(N1),.....MINIMAX(Nm)}
        else
                return min{MINIMAX(N1),.....MINIMAX(Nm)}
end MINIMAX;
```

INPUT :

Input position on the tic-tac-toe board.

EXPECTED OUTPUT :

Human wins Or Computer wins Or Draw .

MATHEMATICAL MODEL :

Let S be the solution perspective .

S={s, e, i, o, f, DD, NDD, success, failure}

s = { Initial state of TIC-TAC-TOE containing blank in all tiles }

I = Input of the system → { I1}
where I1 = { Position on the tic-tac-toe board }

o = Output of the system → { O1 , O2 . O3}
where O1 = { Human wins}
O2 = { Computer wins }
O3 = { Game tie }

f = Functions used → { f1}
where f1 = { Minimax algorithm }
f2 = { Tkinter algorithm }

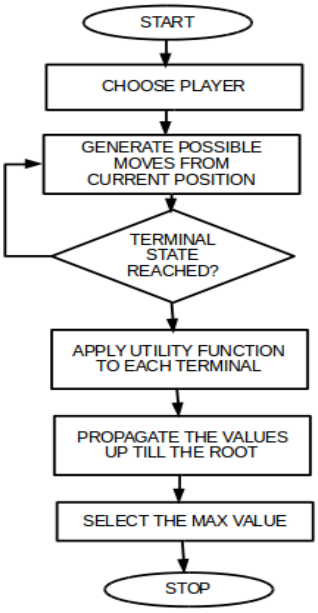
DD - Deterministic data → { }

NDD - Non deterministic data → { Input position, Outcome }

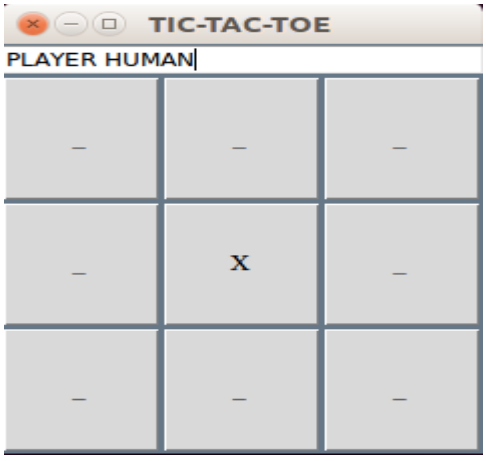
Success - Desired outcome generated → {Unification result }

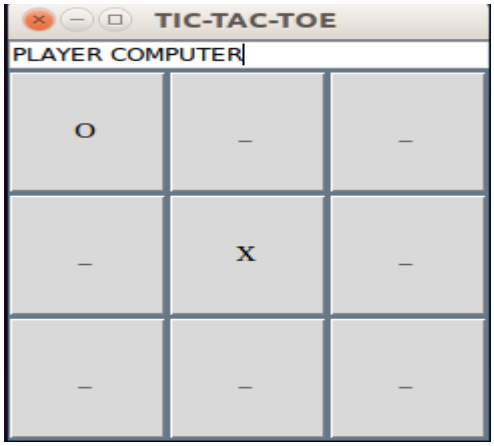
Failure - Desired outcome not generated or forced exit due to system error.

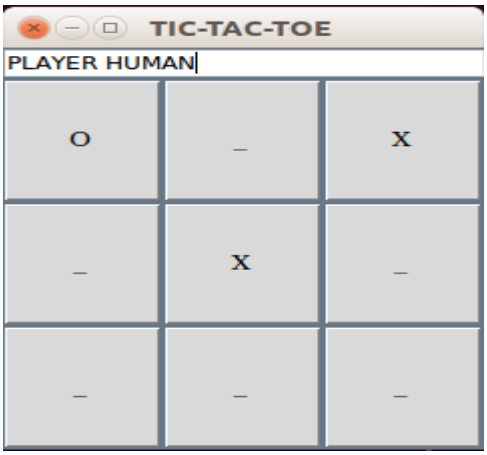
FLOWCHART :

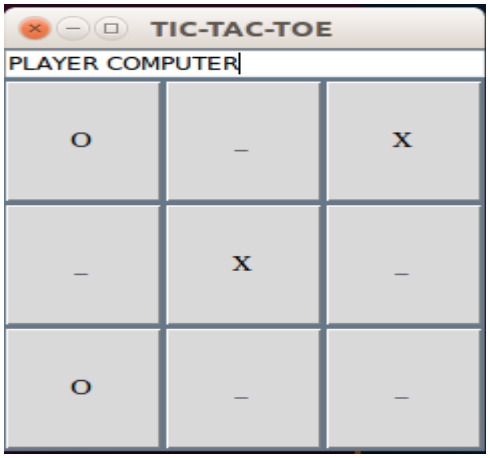


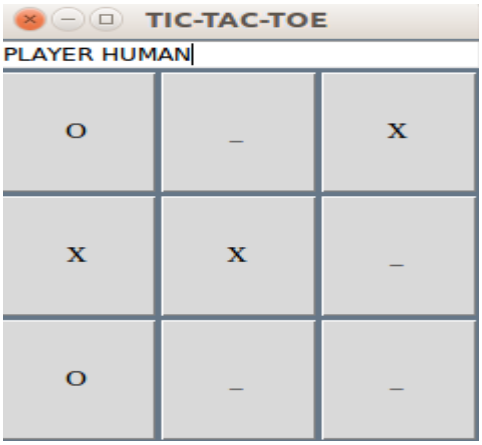
TEST CASES (DRAW CONDITION) :

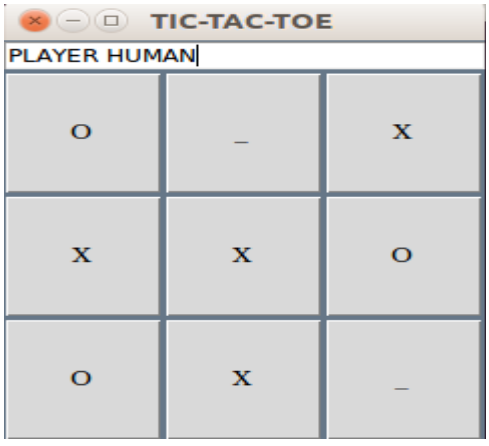


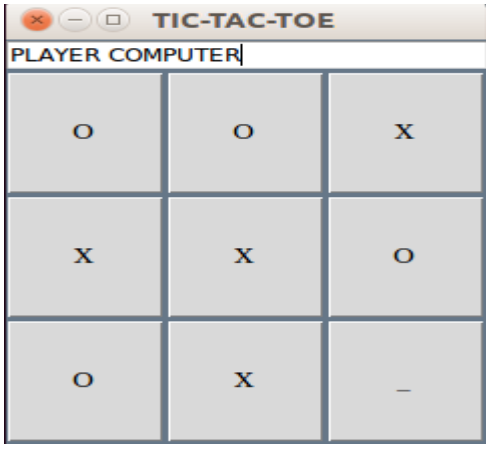


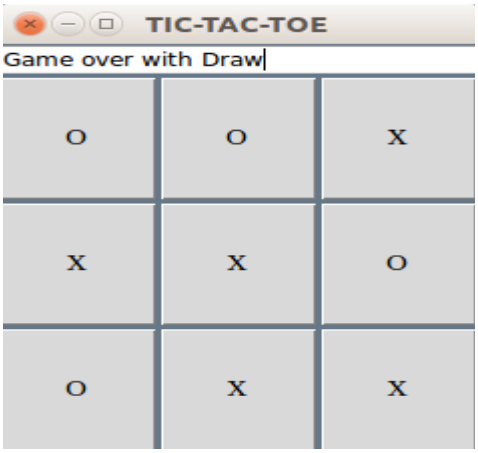




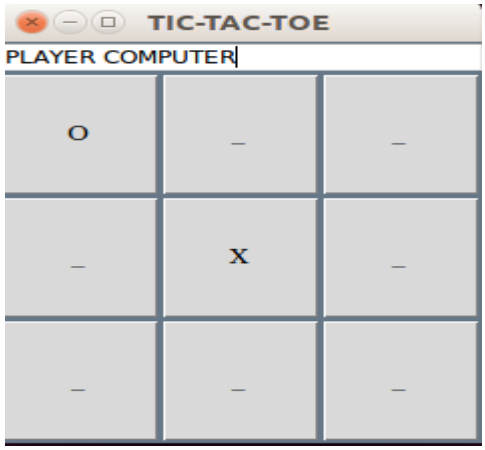


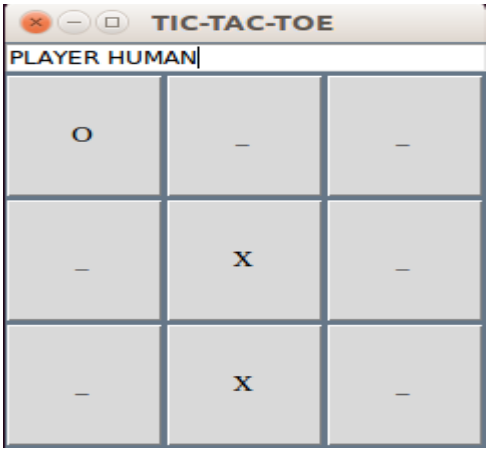


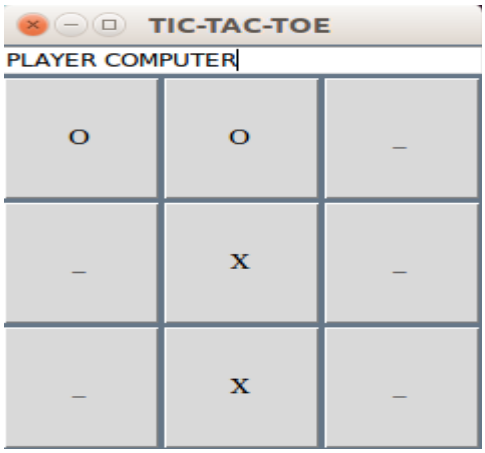


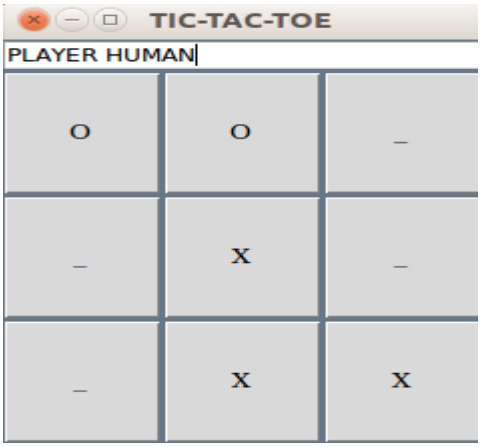


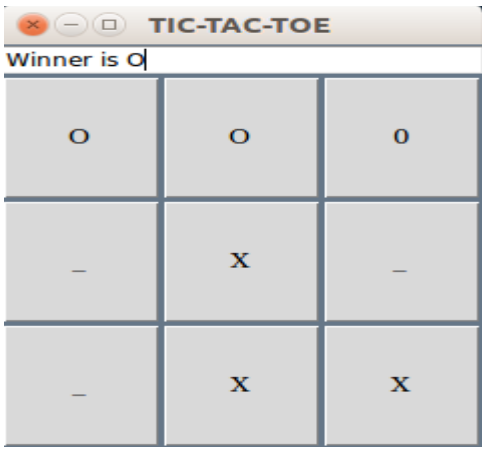
TEST CASES (WINNING CONDITION) :











SPACE AND TIME COMPLEXITIES :

If the maximum depth of the tree is m and there are b legal moves at each point, then the time complexity of the minimax algorithm is O(bm). The space complexity is O(bm.) for an algorithm that generates all actions at once, or O(m) for an algorithm that generates actions one at a time. For real games, of course, the time cost is totally impractical, but this algorithm serves as the basis for the mathematical analysis of games and for more practical algorithms.

CONCLUSION :

Hence we have successfully implemented the minimax algorithm for the tic-tac-toe game.

OUTCOMES ACHIEVED :

COURSE OUTCOME	**ACHIEVED(√)**
Problem solving abilities for smart devices.	
Problem solving abilities for gamifications.	√
Problem solving abilities of pervasiveness,embedded security and NLP.	
To solve problems for multicore or distributed,concurrent/Parallel environments	√