ASSIGNMENT - 6

AIM : Implementation of Mini-Max approach for TIC-TAC-TOE using Java 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 learn and understand Mini-Max searching algorithm.
- To develop Mini-Max approach for TIC-TAC-TOE game.

SOFTWARE REQUIREMENTS :

- Linux Operating System
- Java Compiler
- Eclipse IDE

MATHEMATICAL MODEL :

Consider a following set theory notations related to a program. The mathematical model M for TIC-TAC-TOE is given as below,

M={S,So,A,G}

Where,
S= State space.A state description specifies the location of each of the player(player X and player O) in one of the nine squares.
So= Initial State.Any state can be designated as the initial state.
A= Set of Actions/Operators.The simplest formulation defines the actions as movements of the player X and player O.Different subsets of these are possible depending on where the position of opponent.
G= Goal state., In case of TIC-TAC-TOE user can decide a goal state.

THEORY :

Search algorithms tend to utilize a cause-and-effect concept–the search considers each possible action available to it at a given moment; it then considers its subsequent moves from each of those states, and so on, in an attempt to find **terminal states** which satisfy the **goal conditions** it was given. Upon finding a goal state, it then follows the steps it knows are necessary to achieve that state.

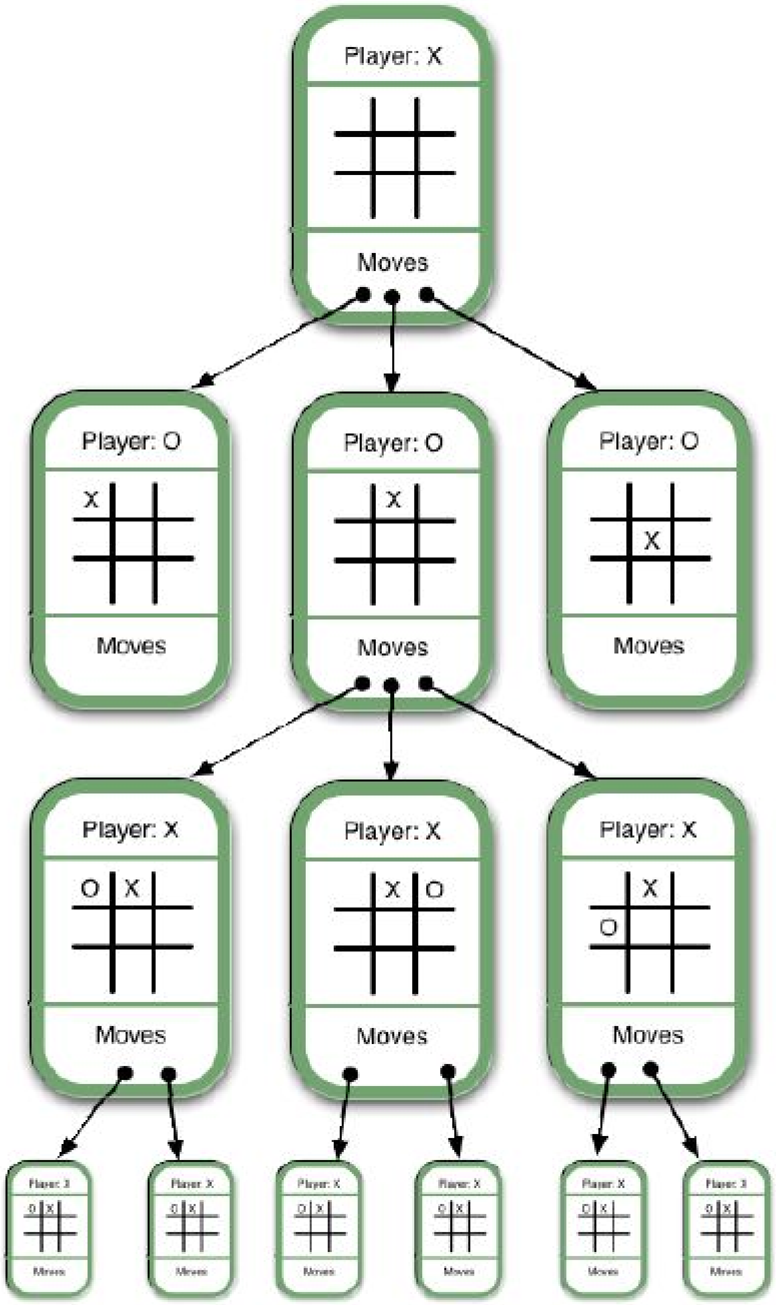
However, in a competitive multi-player game, when other agents are involved and have different goals in mind (and agents usually in fact have goals which directly oppose each other), things get more complicated. Even if a search algorithm can find a goal state, it usually cannot simply take a set of actions which will reach that state, since for every action our algorithm takes towards its goal, the opposing player can take an action which will alter the current state (presumably in a way unfavorable to our algorithm.) This does not mean searches are useless in finding strategies for multi-player games; they simply require additional tactics to be effective.

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.

The Min-Max algorithm is applied in two player games, such as tic-tac-toe, checkers, chess, go, and so on as it helps us to find the accurate values of the board position. All these games have at least one thing in common, they are logic games. This means that they can be described by a set of rules and premises. With them, it is possible to know from a given point in the game, what are the next available moves. So they also share other characteristic, they are full information games. Each player knows everything about the possible moves of the adversary, so we assume that the player will always try to play his/her best move.

The minimax algorithm is used to determine which moves a computer player makes in games like tic-tac-toe, checkers, othello, and chess. These kinds of games are called games of perfect information because it is possible to see all possible moves. A game like scrabble is not a game of perfect information because there's no way to predict your opponent's moves because you can't see his hand.

Representing Moves with the Game Tree Data Structure :


Player: X
Moves
Player: O
Moves
Player: O
Moves
Player: O
Moves
Player: X
Moves
Player: X
Moves
Player: X
Moves

Here's an example of a Game Tree for tic-tac-toe:

Note that this isn't a full game tree. A full game tree has hundreds of thousands of game states, so that's uh... not really possible to include in an image. Some of the discrepancies between the example above and a fully-drawn game tree include:

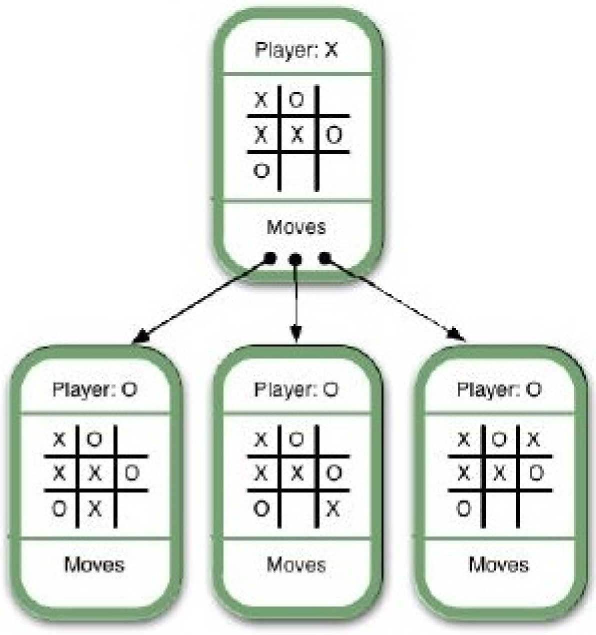
- The first Game State would show nine moves descending from it, one for each of the empty spaces on its board.
- Similarly, the next level of Game States would show eight moves descending from them, and so on for each Game State.

Representing the game as a game tree allows the computer to evaluate each of its current possible moves by determining whether it will ultimately result in a win or a loss. We'll get into how the computer determines this in the next section, Ranking. Before that, though, we need to clearly define the central concepts defining a Game Tree:

- The board state. In this case, where the X's and O's are.
- The current player - the player who will be making the next move.
- The next available moves. For humans, a move involves placing a game token. For the computer, it's a matter of selecting the next game state. As humans, we never say, "I've selected the next game state", but it's useful to think of it that way in order to understand the minimax algorithm.
- The game state - the grouping of the three previous concepts.

  So, a Game Tree is a structure for organizing all possible (legal) game states by the moves which allow you to transition from one game state to the next. This structure is ideal for allowing the computer to evaluate which moves to make because, by traversing the game tree, a computer can easily "foresee" the outcome of a move and thus "decide" whether to take it.

  Next we'll go into detail about how to determine whether a move is good or bad.



Ranking Game States :

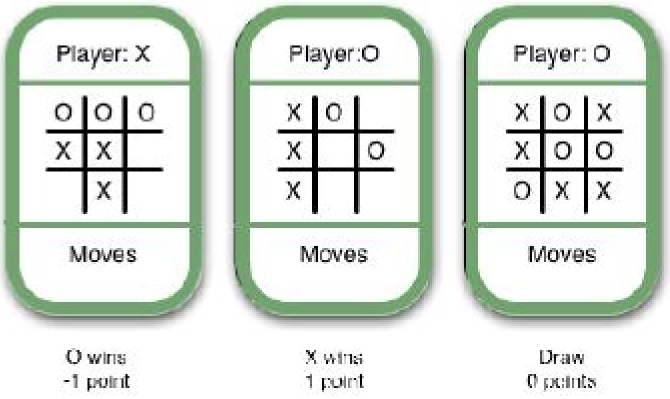
The basic approach is to assign a numerical value to a move based on whether it will result in a win, draw, or loss. We'll begin illustrating this concept by showing how it applies to final game states, then show how to apply it to intermediate game states.

Final Game States :

Have a look at this Game Tree: It's X's turn, and X has three possible moves, one of which (the middle one) will lead immediately to victory. It's obvious that an AI should select the winning move. The way we ensure this is to give each move a numerical value based on its board state. Let's use the following rankings:

- Win: 1
- Draw: 0
- Lose: -1

These rankings are arbitrary. What's important is that winning corresponds to the highest ranking, losing to the lowest, and a draw's between the two.



Since the lowest-ranked moves correspond with the worst outcomes and highest-ranked moves correspond with the best outcomes, we should choose the move with the highest value. This is the "max" part of "minimax". Below are some more examples of final game states and their numerical values:

Intermediate Game States :

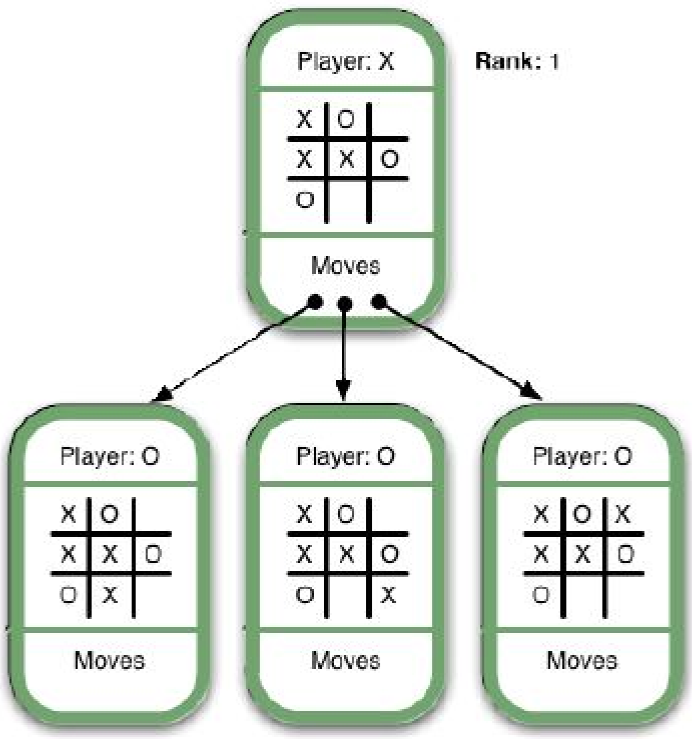
Now have a look at the following game tree:

It's O's turn, and there are 5 possible moves, three of which are shown. One of the moves results in an immediate win for O. From X's perspective this Game State is equivalent to a loss, since it allows O to select a move that will cause X to lose. Therefore, its rank should be -1. In general, we can say that the rank of an intermediate Game State where O is the current player should be set to the minimum rank of the available moves. That's what the "mini" in "minimax" refers to.

By the way - the above game tree probably looks ridiculous to you. You might say, "Well of course you shouldn't make such a dumb move. Why would anyone give up a win and allow O to win?" Minimax is our way of giving the computer the ability to "know" that it's a dumb move, too.

To sum up:

- Final Game States are ranked according to whether they're a win, draw, or loss.
- Intermediate Game States are ranked according to whose turn it is and



the available moves.

- If it's X's turn, set the rank to that of the maximum move available. In other words, if a move will result in a win, X should take it.
- If it's O's turn, set the rank to that of the minimum move available. In other words, If a move will result in a loss, X should avoid it.

ALGORITHM :

```
MinMax(GamePosition game)
{
return MaxMove(game);
}

MaxMove (GamePosition game)
{
```

```
if (GameEnded(game))
{
return EvalGameState(game);
}
else
{
best_move<-{};
moves<-GenerateMoves(game);
ForEach moves
{
move<-MinMove(ApplyMove(game));
if(Value(move)>Value(best_move))
{
best_move<-move;
}
}
return best_move;
}
}

MinMove (GamePosition game)
{
best_move<-{};
moves<-GenerateMoves(game);
ForEach moves
{
move<-MaxMove(ApplyMove(game));
if(Value(move)>Value(best_move))
{
best_move<-move;
}
}

return best_move;
}
```

CONCLUSION :

Thus, we have implemented a Mini-Max approach for TIC-TAC-TOE using Java Eclipse.

Roll No.	Name of Student	Date of Performance	Date of Submission	Sign.
		/ /2015	/ /2015	