Software Architectures
TAKE HOME EXAM

<Adeel Ahmed>: RCSE<WS 2020-2021>

Tick Tac Toe Project Report

Creating A New Player Class

This document describes the implementation of a new player class in a pre-coded Tick Tac Toe game. The board size is 4x4 and one box is randomly blocked at the start of the game. The players may also be switched at any time of the game. Currently with the provided code there is a CPUvsCPU option where two CPU players play randomly with each other.

The Task

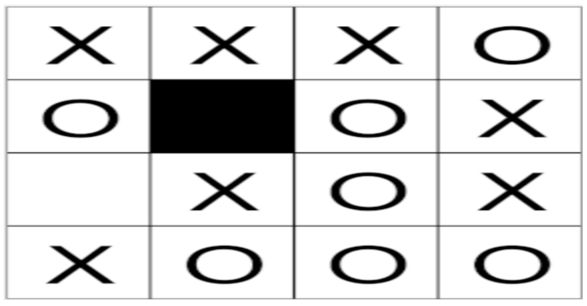
We are provided with the a 4x4 board as an integer array we are able to access all the 16 fields and check the data and can select any of the empty fields and place the mark *(X/O)*.

We need to make a strategy to beat other player objects. We are allowed to inherit from the Player class and our entry point in the game is *getNextMove*. The framework will call this method whenever our player class is asked to compute the next move.

My Solution

First of all, we needed to make a new class and then link it up with the other premade classes. Our entry point in the game is GetNextMove so what the code does is that it checks the player number first to see who is playing then the code checks if the spots are available on the board. If a spot is available the code calls the MiniMax Algorithm. The Minimax Algorithm in return gives us the optimum move to play. The move is made and then the code checks for a line to see if anyone has won. If there is no winner and spots are empty the game continues till all the spots are filled.

Figure 1: Our Tick Tack Toe Board



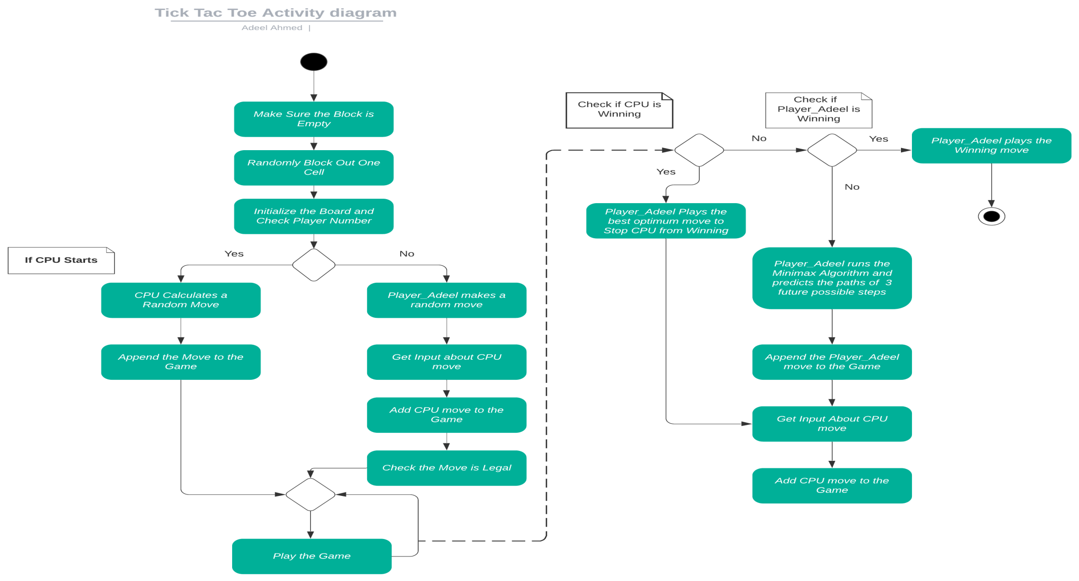
Software Architecture of Player_Adeel

Let us discuss the intelligent algorithm that will play *Tick Tac Toe* and will not lose a game.

The Activity Diagram

To Understand our Architecture better a UML activity diagram has been made. This diagram will give us brief idea how the game is being played.

Figure 2: UML Activity Diagram


Tick Tac Toe Activity diagram
Adeel Ahmed |
Make Sure the Block is Empty
Randomly Block Out One Cell
Initialize the Board and Check Player Number
If CPU Starts
Yes
No
CPU Calculates a Random Move
Player_Adeel makes a random move
Append the Move to the Game
Get Input about CPU move
Add CPU move to the Game
Check the Move is Legal
Play the Game
Check if CPU is Winning
Check if Player_Adeel is Winning
No
Yes
Player_Adeel plays the Winning move
Yes
No
Player_Adeel Plays the best optimum move to Stop CPU from Winning
Player_Adeel runs the Minimax Algorithm and predicts the paths of 3 future possible steps
Append the Player_Adeel move to the Game
Get Input About CPU move
Add CPU move to the Game

Code Design

First of all we assign the developer name and player_number. The player_number has very significance in our scenario as it will tell us who is playing either the player or the computer. As it is quite likely that the player turns be reversed in game we need to know who is taking turns.

Then we move into *GetNextMove*:

```
int Player_adeel::getNextMove(int fld_status[])
```

where we check the player number check either the fields are available and keep a score. Then we call the MiniMax function which computes us the best score and the best move is played. We will discuss the implementation of Algorithm in later.

Later what we needed to compute *getResultsIfGameOver* the function shown:

```
Player_adeel::Result Player_adeel::getResultIfGameOver(int fld_status[])
```

This is an important part of code. We need to know either someone won the game. To check the Diagonals we see if the respective diagonal field places are equal or not. For the Horizontal and Vertical a simple *for* loop is used which checks the field status and returns the result. Then we just simply check if the Values are equal and simply return it back.

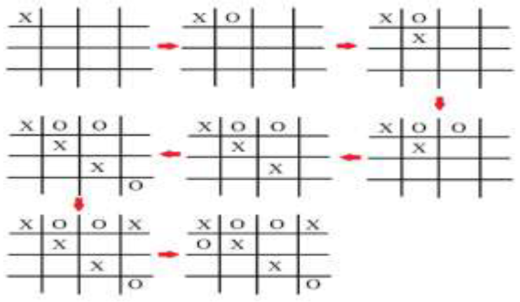
MiniMax Algorithm

The MiniMax Algorithm is the brain behind the computation of next move. The MiniMax function in the code is written as:

```
int Player_adeel::runMiniMax(int fld_status[], int depth, bool isMaximizing)
```

Let us Understand the MiniMax function briefly. This is the algorithm that enables the player to look forward to future moves and pick the move that gives him the best alternatives for future moves. It has two fundamental capacities, the Min-Move and Max-Move capacities, which are commonly recursive. Given a lot of potential moves, the player thinks about what the adversary's best move would be in every one of those potential states and after that picks the move that gives his rival the most exceedingly awful choices. Additionally, the Min-Move capacity considers all the

Figure 3: 4x4 MinMax Algorithm Working



potential moves from a given state, thinks about what the player's best move would be from those, and returns the estimation of his most exceedingly terrible choice. This common recursion proceeds until as far as possible is come to, so, all in all the capacity just assesses all the potential states.

MiniMax Implementation in Code

The pseudocode for our the depth limited (will discuss depth later) minimax algorithm is given below:

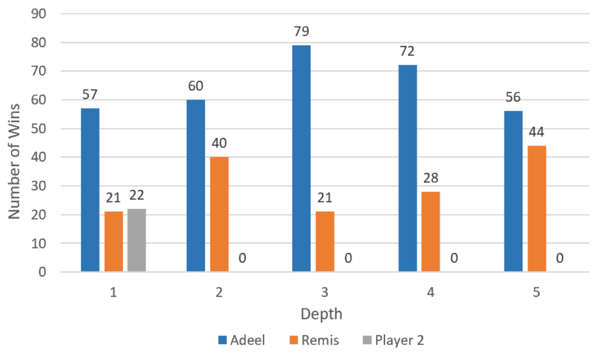
```
function minimax(node, depth, maximizingPlayer) is
    if depth = 0 or node is a terminal node then
        return the heuristic value of node
    if maximizingPlayer then
        value := − Infinity
        for each child of node do
            value := max(value, minimax(child, depth minus 1 , FALSE))
        return value
    else (* minimizing player *)
        value := +infinity
        for each child of node do
            value := min(value, minimax(child, depth minus 1, TRUE))
        return value
```

Let us briefly discuss the code, we start with initialzing our node that in this case is fld_status and then we initialize the depth and a bool to store the Maximizing unit. We start with the getting the result and to do that we have to make use of an enum class. I tried to create simple enum but visual studio context did not allowed me and asked to make a class. Then we checked the maximizing player similarly checking the availability of spot. The algorithm will run recursively and get us the best score by going through the Maximizing Player or minimizing player depending on the available condition. The Algorithm is quite simple to implement.

Results and Shortcomings of MiniMax

Although MiniMax is a quite good Algorithm but it is does not perform very well for 4x4 boxes as the number of possible moves are 16! which is = 20,922,789,888,000 iterations and it takes too much time. So in order to avoid this we have to set the value of depth. It has been observed that after depth 6 the output gets very slow. So i have tested around 500 games with depth 1 to 5 to see which one is optimum for our game setup and plotted the graph shown: So the graph shows

Figure 4: Wins Vs Depth



that we have the best result at depth 3. So i have chosen the depth cutoff in the code as 3 and it works fine. After doing this test cases are made and i have tried to cover all of cases. I donnot have a tool to check my coverage but i think they have 100% coverage.

Future Prospects

Although the code works fine by placing a depth cutoff but the solution is not optimum and a better way to implement this code is implementing Alpha-Beta Pruning Algorithm which takes care of this problem. But due to lack of time and being in Quarantine i was not able to add that to my code. Hopefully will implement this in future.

Reference

Figure 2 Image has been taken from Google Images and MinMax Algorithm Pseudocode has been taken from Wikipedia for Educational Purposes under fair use doctrine.