CS 640 Programming Assignment 3 Report

AI Game —— 4×4×4 Tic-Tac-Toe

November 24, 2019

Teamwork

Team members	Kerberos	BU ID	Works
Ziqi Tan	ziqi1756	U 88387934	Design data structure and algorithm. Coding.
Kaijia You	caydenyo	U 44518396	Coding and software testing. Write documentations.
Tian Ding	dingtian	U 90706530	Coding. Adjusting parameters.

The rest of this report is organized as follows. First, we review the assignment requirements. Second, we provide insight into the 4×4×4 Tic-Tac-Toe Game mostly based on [1]. Third, we go through the skeleton code and discuss our methodology. Finally, we discuss our test result. Our strategy can easily defeat the algorithm with simple defend and attack strategy and perfectly defeat random algorithm.

Assignment Requirements

In this assignment, we are required to implement an AI 4x4x4 cubic tic-tac-toe game by using minimax and alpha-beta pruning method which drive our AI make decisions as beneficial as possible. We should try our best to modify our algorithm and beat AI implemented from other teams for extra credits.

Insight into 4×4×4 Tic-Tac-Toe Game

According to [1], the first player in 4×4×4 Tic-Tac-Toe Game can always force a win. Additionally, no draw exists.

Skeleton Code Quick Review

runTicTacToe.java serves as the game engine. The **run()** method is a critical part. It should be recognized that the decision algorithm (myAIAlgorithm(board, player)) will be called many times in the game.

```java
public void run()
{

    Random rand = new Random();
    int turn = rand.nextInt(2)+1; //1 = player1's turn, 2 = player2's turn, who go first is randomized

    while((result = isEnded())==0) //game loop
    {
        if(turn==1)
        {
            positionTicTacToe player1NextMove = ai1.myAIAlgorithm(board,1); //1 stands for player 1
            if(makeMove(player1NextMove,1,board))
                turn = 2;
        }
        else if(turn==2)
        {
            positionTicTacToe player2NextMove = ai2.myAIAlgorithm(board,2); //2 stands for player 2
            if(makeMove(player2NextMove,2,board))
                turn = 1;
        }
        else
        {
            //exception occurs, stop
            System.out.println("Error!");
        }
    }
```

Data structure

```java
public class positionTicTacToe {

    int x;
    int y;
    int z;
    int state;
    // mark by player 1 or player 2 or null ('X', 'O', '_')
    // { 0 : unmarked, -1 : No meaning, 1 : 'X', 2: 'O' }

    /**
     * Method: printPosition
     * Function: print positions(x, y, z) and state.
     * @author TF in CS 640 at Boston University
     * */
    public void printPosition() {
        System.out.print("("+x+","+y+","+z+")");
        System.out.println("state: " + state);
    }

    /**
     * Constructor
     * @author TF in CS 640 at Boston University
     * */
    positionTicTacToe(int setX, int setY, int setZ, int setState) {
        x = setX;
        y = setY;
        z = setZ;
        state = setState;
    }

    /**
     * Constructor
     * @author TF in CS 640 at Boston University
     * */
    positionTicTacToe(int setX, int setY, int setZ) {
        x = setX;
        y = setY;
        z = setZ;
        state = -1;
    }
}
```

Methodology

1. Evaluation Function

In every board configuration, the player should evaluate his/her current situation, which is a crucial part of a heuristic process for an Artificial Intelligence. We adapt the following strategy [2].

The evaluator has a list of how many 1-in-a-rows, 2-in-a-rows, 3-in-a-rows and 4-in-a-rows each player has. Where the evaluators differ is in what they do with this information, as described below.

```
private int[] playerSequenceNum = new int[4];
private int[] opponentSequenceNum = new int[4];
```

It assigns a positive value to every n-in-a-row the player has, and a negative value to every n-in-a-row the opponent has. An n-in-a-row is worth about an order of magnitude more than an (n-1)-in-a-row.

```
private static byte[][] winningLine = new byte[76][4];
private static final int[] playerSequenceValue = new int[] {1, 15, 130, 100000};
private static final int[] opponentSequenceValue = new int[] {-1, -10, -100, -100000};

  int value = 0;
  for( int i = 0; i < 4; i++ ) {
      value += playerSequenceNum[i] * playerSequenceValue[i];
      value += opponentSequenceNum[i] * opponentSequenceValue[i];
  }
```

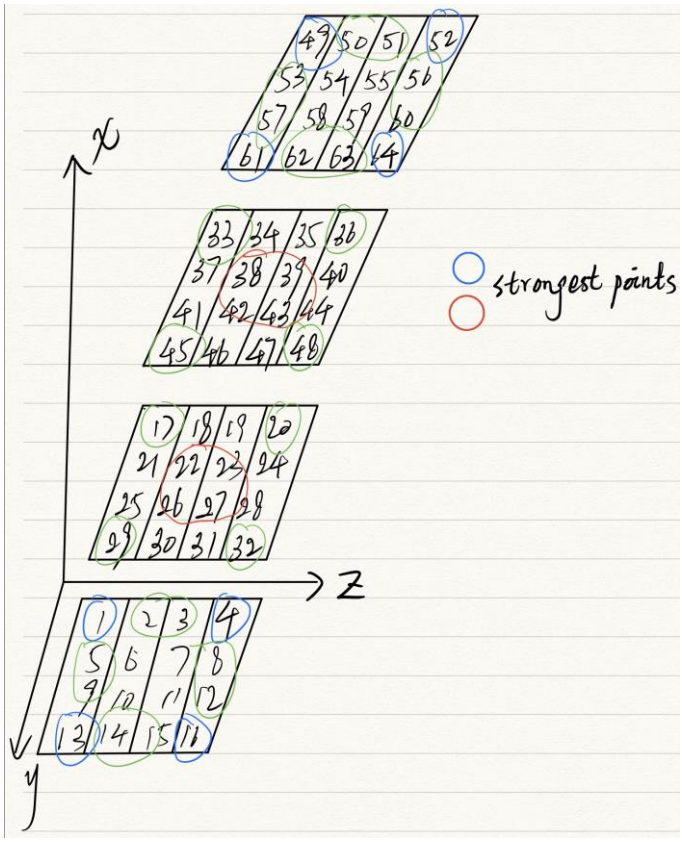
2. Change Data Structure

```
private byte[] curBoard = new byte[64];
```

Without deep copy of a list of objects, this board data structure takes much less time and memory.

In the skeleton code, position in the cube is represented by an object that contains x, y, z and status which occupies a lot of memory to store when the scale of the cube becomes larger. What's more, the for-loop in the makeMove method is terribly time cost and not necessarily. We have already known the exact node that we want to mark after doing minimax and alpha-beta pruning, however, the board is store in a List that means we must iterate the items in the List to get the position we want, which will cost O(n) in the worst case. The efficiency will be obviously better if the board is store in an array, we can mark the position simply by the subscript of the array and it only takes O(1). For this reason, we came up with the version 2 of this design.

3. **Occupy the strongest nodes as fast as possible**



The strongest point is the point that cross by most of the winning lines. The strongest points shown in the picture has 8 wining lines respectively. It is easy to understand that the 8 nodes in the center of the cube give players more winning strategies. Each player will not win in the beginning of the game, so it is important for players to occupy the strongest nodes as fast as possible when the game begins.

4. **Winning move and force move**

As the game begins, the board will be different after players finish their turns. Player will check if 3 nodes of his/her own are filled in the same line, if yes, then player can take a winning move immediately. Otherwise player should check if 3 nodes of opponent are filled in the same line, is yes, then player has no choice but have to stop opponent from winning which is called force move.

5. **minimax**

If the current chess board is not fit in the situation above then AI should make decisions by minimax and alpha-beta pruning method to maximize the its profit and minimize the opponent's profit at the same time.

```
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 := −∞
        for each child of node do
            value := max(value, minimax(child, depth − 1, FALSE))
        return value
    else (* minimizing player *)
        value := +∞
        for each child of node do
            value := min(value, minimax(child, depth − 1, TRUE))
        return value
```

Pseudocode of minimax

6. Alpha-beta pruning

```
function alphabeta(node, depth, α, β, maximizingPlayer) is
    if depth = 0 or node is a terminal node then
        return the heuristic value of node
    if maximizingPlayer then
        value := −∞
        for each child of node do
            value := max(value, alphabeta(child, depth − 1, α, β, FALSE))
            α := max(α, value)
            if α ≥ β then
                break (* β cut-off *)
        return value
    else
        value := +∞
        for each child of node do
            value := min(value, alphabeta(child, depth − 1, α, β, TRUE))
            β := min(β, value)
            if α ≥ β then
                break (* α cut-off *)
        return value
```

Pseudocode of alpha-beta pruning

7. Progressive Deepening

Analyze game situation to depth = 1, depth = 2, depth = 3, ··· until time is up.

8. Heuristic Pruning

Define the traverse order. We traverse the strongest points first, which are obviously promising move.

```
private static final int[] traverseOrder = {
    21, 22, 25, 26, 37, 38, 41, 42,
    0, 3, 12, 16, 48, 51, 60, 63,
    1, 2, 4, 5, 6, 7, 8, 9,
    10, 11, 13, 14, 15, 17, 18, 19,
    20, 23, 24, 27, 28, 29,
    30, 31, 32, 33, 34, 35, 36, 39,
    40, 43, 44, 45, 46, 47, 49,
    50, 52, 53, 54, 55, 56, 57, 58, 59,
    61, 62
};
```

Algorithm Design

1. Pseudocode of myAIAlgorithm

Algorithm 1 myAIAlgorithm

Input List<positionTicTacToe> *board*, int *player*
Return positionTicTacToe *myNextMove*

procedure MYAIALGORITHM(*board*, *player*)

initialization

winMove ← *getWinMove*(*player*) ▷ If we have a win move
if *winMove* exists **then**
  return *winMove*

forceMove ← *getForceMove*(*player*) ▷ If we have a force move
if *forceMove* exists **then**
  return *forceMove*

coreMove ← *getFirstTwoSteps*(*player*) ▷ Occupy the strongest points
if *coreMove* exists **then**
  return *coreMove*

maxValue ← $-\infty$
positionTicTacToe myNextMove

do ▷ Progressive deepening
  for `<each available move` *curMove*`>` **do**
    `<make current move>`
    newVale ← *miniMax*(*depth*, *player*, *false*, $-\infty$, $+\infty$)
    if *newValue* > *maxValue* **then**
      maxValue ← *newValue*
      myNextMove ← *curMove*
    `<cancel current move>` ▷ Backtracking
while `<time is still enough>`
return *myNextMove*

2. Pseudocode of miniMax

Algorithm 2 miniMax

```
Input int depth, int player, boolean maximizingPlayer, int alpha, int beta
Return int value
procedure MINIMAX
    if depth == 0 then                                          ▷ search finish
        return evaluation(player)                ▷ evaluate the board configutaion

    if maximizingPlayer then                                    ▷ Maximizer
        value ← −∞
        for <each available move curMove> do
            winMove ← getWinMove(player)                  ▷ win move pruning
            if winMove exists then
                <make this win move>
                value ← evaluation(player)
                <cancel this win move>                          ▷ Backtracking
                break
            forceMove ← getForceMove(player)              ▷ force move pruning
            if forceMove exists then
                <make this force move>                    ▷ DFS and deepening
                value ← max(value, miniMax(depth, player, false, alpha, beta)
                <cancel this force move>                        ▷ Backtracking
            else                                                ▷ Naive miniMax
                <make current move>                                   ▷ DFS
                value ←
                max(value, miniMax(depth − 1, player, false, alpha, beta))
                <cancel current move>                           ▷ Backtracking
                alpha ← max(alpha, value)
                if alpha >= beta then
                    break
        return value
    else                                                        ▷ Minimizer

        value ← +∞
        opponent ←!player
        for <each available move curMove> do
            winMove ← getWinMove(opponent)                ▷ win move pruning
            if winMove exists then
                <make this win move>
                value ← evaluation(player)
                <cancel this win move>                          ▷ Backtracking
                break
            forceMove ← getForceMove(opponent)            ▷ force move pruning
            if forceMove exists then
                <make this force move>                    ▷ DFS and deepening
                value ← min(value, miniMax(depth, player, true, alpha, beta))
                <cancel this force move>                        ▷ Backtracking
            else                                                ▷ Naive miniMax
                <make current move>                                   ▷ DFS
                value ←
                min(value, miniMax(depth − 1, player, true, alpha, beta))
                <cancel current move>                           ▷ Backtracking
                beta ← min(beta, value)
                if alpha >= beta then
                    break
        return value
```

Results

1. **Battle between final AI version and original AI version.**

   Result: final version always wins.

```
Player1' turn:
Evaluation 1789
Progressive deepening: depth = 4
Progressive deepening: depth = 5
Progressive deepening: depth = 6
Progressive deepening: depth = 7
Progressive deepening: depth = 8
Progressive deepening: depth = 9
myNextMove: 2 1 2 Value: 101902
Player 1 run time: 9093 ms
level(z) 0
[O_XX]
[XOXX]
[XOXX]
[X_OO]

level(z) 1
[O_XO]
[_OXO]
[_OO_]
[OXXX]

level(z) 2
[OOOX]
[OXXX]
[_XOX]
[__OO]

level(z) 3
[XOXX]
[O_OX]
[__OO]
[___X]

(0,3,2)state: 1
(1,2,2)state: 1
(2,1,2)state: 1
(3,0,2)state: 1
Player1 Wins
level(z) 0
[O_XX]
[XOXX]
[XOXX]
[X_OO]

level(z) 1
[O_XO]
[_OXO]
[_OO_]
[OXXX]

level(z) 2
[OOOX]
[OXXX]
[_XOX]
[X_OO]

level(z) 3
[XOXX]
[O_OX]
[__OO]
[O__X]

Program run time: 188395 ms
Player 1 is the first player.
player1: 10 player2: 0
```

   Where player 1 is the final version of AI and player 2 is our original version.

   No mater who is the first player, final version always wins.

2. **Battle between final version as a first player and final version as a second player.**

```
Player 1 is the first player.
player1: 3 player2: 7
```

   Result: the second player performs better than the first player.

   The reason may be that

   - the second player has much more time to do progressive deepening as the number of available moves gets less and less.
   - Besides, the heuristic value is not always correct.

```
private static final int[] playerSequenceValue = new int[] {1, 15, 130, 100000};
private static final int[] opponentSequenceValue = new int[] {-1, -10, -100, -100000};
```

   - Additionally, in the first move of each player, they randomly choose one strongest point in the center of the cubic, which means the first step is extremely important for each player in this game.

References:

[1] Patashnik, Oren. "**Qubic: 4 × 4 × 4 Tic-Tac-Toe**." Mathematics Magazine 53, no. 4 (1980): 202-16. doi:10.2307/2689613.

[2] **3D Tic Tac Toe Algorithms** - Rochester CS
https://www.cs.rochester.edu/u/brown/242/assts/studprojs/ttt10.pdf