Project 1 TTT Game & Minmax Algorithm Report

Runhua Gao

Overview:

The whole Project is to implement the **MinMax algorithm** and **Heuristic Function** (also called A-star algorithm) in a state space. The state is current status of a Tic-Tac-Toe game process.

To make the game go smoothly, **an easy implemented API design** really matters. I design mainly two API

Node API

- *String state(): return the string representation of current node*
- *Int size(): return the basic size*
- *List<Integer> actions: possible actions(empty positions) of current Node*
- *List<char[]> lines: the line in char which may end the Game(row, column, diagonal lines)*
- *Void transition(int board(), int position): the transition model for the node*
- *Turn(): return the round of current node(it is player's round or AI's round)*
- *checkIsTerminal(): check if this node is a terminal node(specify for each type of node)*
- *IsTerminal():return if the node is a termianl node*
- *Utility(): if the node is a terminal node,return it's utility value(-1,0,+1)(could only call this method when it is a terminal node,else would cause exception)*
- *HValue(): only used for Part II & Part III's node, return it's heuristic value*
- *Showstate(): the format of board will be printed on the terminal*
- *ValidMove(int board, int position): validate if the move if valid(used for transition modeling)*

Game API

- *Game(): single game process*
- *UpdatePlayerRound(): update current state in player's round*
- *UpdateAIRound():update current state in AI's round*
- *ReadPlayerTurn(): ask the player which pawn he/she wants to use*
- *ReadPlayerMove(): read the player's move and clarify if they are valid*
- *EndGame(): when the current state is terminal, clarify who wins the game*

These two standard API are the structure of whole game process, the details of implementation of algorithms are described in later sections. TTTGame has some different named methods from the standard API but actually executing same function.

The **HMinMax class** is to implement the HMinMax algorithm,(basic minmax algorithm for part is just a private class in basic game class because it has totally different structure from the HMinMax), the **HeuristicAdvanNode** and **HeuristicQubicNode** are both the class to **calculate heurstic value** for AdvanNode and QubicNode, respectively. **Gameparameters class**(a static class) is to **provide some standard parameters** for each Node class and Game process(like flag x and o).

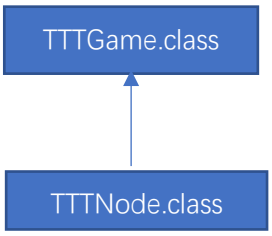
Description of source files:

- *(TTTNode.class, TTTGame.class): Node and Game Process for Part 1*
- *(AdvanNode.class, Advan3TGame.class, HeuristicAdvanNode): Node, heuristic value, Game Process for Part 2*
- *(QubicNode.class, QubicGame, HeurisQubicNode): Node, heuristic value, Game process for part 3*
- *HMinMax.class: implementation of HMinMax and alpha beta pruning for Part 2 & Part3*
- *GameParameters.class: some standard parameters used for whole project(like x & o)*

Structure(Hierarchy) of project:

Part I:

TTTnode as the state for TTTGame process, TTTGame process has the method to calculate utility and decide to move in which direction in AI's round.

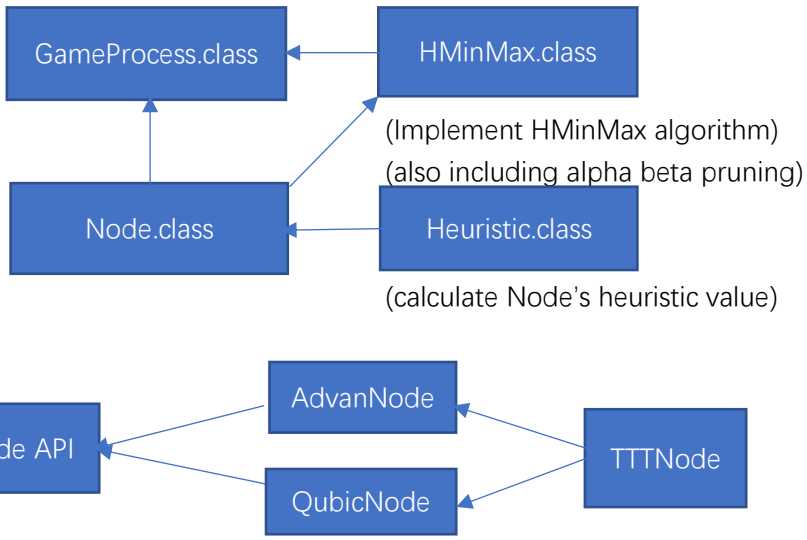


(as the state of the state space of a game)

The TTTNode also implements the node API above except for hVaul method because I do not implement heuristic function in this part.

Part II & Part III:

Because part 2 and part 3 has a same structure, so the hierarchies for these two parts are same.



The AdvanNode and QubicNode strictly implement the Node API, the TTTNode implements most method in the API and constitutes the board(a TTTNode array of ADvanNode and QubicNode).

The arrow start from Node to HMinMax means that when implementing the hMinMax algorithm, if it needs to calculate the heuristic value of a node, just use the method in API(hValue, return an Integer) rather than create a new Heurstic class to calculate because for each type of Node, the rule and class to calculate its heuristic value are totally different. In my project, HeursticAdvanNode is used to calculate value for Part II, the HeursticQubicNode is used to calculate for Part III.(Here I know that a good API design reallly matters again!).

Part I Description:

Nothing special for part I. Basic rule for the game process and basic step to implement the MinMax algorithm. But some details are still need to be discussed.

At the beginning, I assume that because the whole state space has finite number of nodes(states, a smaller number compared to Part II and Part III), I decide to produce all nodes as an instance variable of a game, to speed up when AI algorithm decide to pick which node as next node. But the problem here is that in which game process **it is impossible to reach each status**, so it is a **waste of both space and time** to **preproduce all nodes**. In stead, I should use a strategy called dynamic search, which is to produce nodes only if the game or MinMax search reach it leading a save for both space and time.

I do not use heuristic function and alpha beta for this part because it really has a small number of nodes compared to later section, in which case I could do an integrated search in whole state space to get an optimal solution. Another reason is that each node has only three possible value:-1,0,+1, which do not need a heuristic value and alpha beat pruning.

The **calutility** method in TTTGame.class is a recursive method the calculate the utility value for a node. The point here needs to be discussed is that if we **should immediately stop search when we come across a –1 or +1 node** at some position(this is something like alpha beat pruning), because in next MinMax algorithm a common situation is that we find for example four +1 successor nodes, two 0 successor nodes and one –1 node,. What we need to choose a max value node(for max value search), so as long as we come across a +1 node, we should return immedately (no need to search more nodes).

```
public class TTTGame {
    private final int size;
    private TTTNode currentNode;

    // the constructor
    public TTTGame(int size) { this.size = size; }

    // Game process
    public void gameStart() {...}

    // make sure the turn that the player want to select
    static int playerTurn() {...}

    // make sure the position that player wanna drop the flag is valid
    private int playerMove() {...}

    // calculate the utility value of a node
    private int calUtility(TTTNode node) {...}

    // update current state to next state in player turn
    private void updateStatePlayer(int position) {...}

    // calculate optimal path for the program's turn
    private class calNextMoveProgram {...}
```

Game Process.class for Part 1, the calUtility and calNextMoveProgram is the mehod to be used in AI's round.

Part II & Part III description

Overview:

As described in Hierarchy ,Part II and Part III are totally same in game process, node constitution. The only different point is the way to the calculate their heuristic value and the rule to clarify if a node is a terminal node.

However I wanna discuss something special for Part 2 here, as **it is a neat variant of Part 1**.Based on the special rule of Part 2, we could easily find the successor nodes of the current node is just the successor nodes we get from part1. For example, if we first mark at x at (board = 7,position = 1), the successor nodes are just the successors of board 1 plus other empty nodes(which constitute the AdvanNode), in this case, what we only have to change is just the rule to check if the node is terminal and add Heuristic value calculation used in HMinMax search algorithm.

For these two parts, I use a node structure called **board**: **an array of TTTNode**(the basic node for part I, TTTNode [ ]). For example, the length of board in Part II is 9 and 4 for Part III.

The rule to check if a node is terminal:

For Part Two, just check if each TTTNode in board is a terminal node and if all TTTNodes in board have zero actions(no empty positions) to mark more pawns(in this case it is a draw).

For Part Three, check if each TTTNode in board is a terminal node, vertical lines and diagonal lines in vertical plane(**has a total number of 76 lines to check**) and also, check if this is draw node by checking if there are no empty positions in each TTTNode in the board.

The rule to calculate heuristic value

Part II: As I learn from the lecture, the requirement of a good heuristic evaluated function is to admissible(never overestimate the value) and only based on current state's situation, do not do more search. I design the rules: For each **line** (**the line here is the column, row and diagonal line of each TTTNode in AdvanNode's board**) "**grade**" it.

For example, if a line **has both x and o flags**, means that this line **could never lead a victory** for both player and AI, so we give it a draw point(0).

If a line **has only x or o flag**, then we could **give it a non-zero point** based on its number flags. The points here are totally decided by ourselves, but the requirement is to never overestimate a node(For example, the point of x wins is 1000, but you evaluate a node for a point of 1500). The evaluate point I use here is: 1 for only one x in a line, ten for two x in a line and 1000 for three(the victory). Vice versa.

Some Terminologies:

One Flag Line: has only one x or o flag in a line, Give it a 1(for x) or -1(for o)
For example: a line of the Node is [x,blank,blank] or [o, blank,blank]

Two Flag Line: has only two x or o flags in a line, give it a 10(for x) or -10(for o)
For example: a line of the Node is [x,x,blank] or [o,o,blank]

Three Flag Line: has only three x or o in a line, give it a 1000(for x) or -1000(for o).
1000 and -1000 is used just in Part 2, in Part 3 the point is reduced to +100 and

-100.

For example: a line of Node is [x,x,x] or [o,o,o]

Four Flag Line: has only four x or o in a line, give it a 8000(for x) or -8000(for x)
Only used in Part 3
For example: [x,x,x,x] or [o,o,o,o]
Victory Point: the point that represents a victory for player or AI algorithm.(1000 for part 2 and 8000 for part 3)

```
// calculate the HeuristicAdvanNode's value according to its current state
public class HeuristicAdvanNode {
    //    private static final int draw = 0;
    private static final int twoPoints = 10; // a line that only has two pawns of same type
    private static final int onePoint = 1; // a line that only has one one pawn
    private static final int threePoint = 1000;//already win
```

(standard grade in HeuristicAdvanNode.class)

```
public class HeuristicQubicNode extends HeuristicAdvanNode {
    private static final int xWin = 8000;
    private static final int oWin = -xWin;
    private static final int onePoint = 1;
    private static final int twoPoint = 10;
    private static final int threePoint = 100; // max is 76*100 - 7600 < xWin
```

(standard grade in HeuristicQubicNode.class)

For the calculation of heuristic value of QubicNode, just **reduce** the grade for a three flag Line to 100, **add the victory point of a Four flag line** with 8000.

```
protected int calHValue(int xNum, int oNum) {
    if (xNum > 0 && oNum > 0) {
        return GameParameters.draw;
    } else {
        switch (xNum + oNum) {
            case 1:
                return xNum > 0 ? onePoint : -onePoint;
            case 2:
                return xNum > 0 ? twoPoints : -twoPoints;
            case 3:
                return xNum > 0 ? threePoint : -threePoint;
            default:
                return GameParameters.draw;
        }
    }
}
```

(the xNum and oNum stand for the number of x and o in a line)

The key here is that even though a board has full two flag lines, the number is 10 * 9 = 90(each board has 10 lines), the point we get is 90*10 = 900, still less than a victory point.

But what if there **exists a three flag line**, which may lead to a **point larger than victory point**(1000) or **less than defeat point**(-1000)? The strategy I use here is when return the heuristic value for a Node, check **if the absolute value of point is larger than 1000**. It the point is beyond the range, just return the victory point.

```
private static final int threePoint = 1000;//already win
protected int size = 0;
protected AdvanNode node;
protected int value = 0;

// the constructor
public HeuristicAdvanNode(AdvanNode advanNode) {...}

// return the a AdvanNode's heuristic value
public int heuristic() {
    if (Math.abs(value) > threePoint) {
        return value > 0 ? threePoint : -threePoint;
    }
    return value;
}
```

The above screen shot is some code of my HeuristicAdvanNode.calss. In which you can see when the public method heuristic wants to return an Integer, it **must first clarify if the value**(the sum of point of all lines in a AdvanNode) is beyond the range to **make sure the heuristic value** we get is **admissible**.

The theory for QubicNode is same, even if it is full of 3 flag lines, the heuristic we get is 76*100 =7600, still less than 8000,which make the value admissible.

If there exists a four flag line([x,x,x,x] or [o,o,o,o]) of the node, the value we get may exceeds the victory grade(8000). But the heuristic method in Heuristic value also bound the heuristic value we get.

```
@Override
// return a total heuristic value
public int heuristic() {
    if (Math.abs(value) > xWin) {
        return value > 0 ? xWin : oWin;
    }
    return value;
}
```

(heuristic value method in HeuristicQubicNode.class, xWin here is 8000, oWin is -8000).

In short, all these codes are to make the heuristic value we use in HMinMax search algorithm is **admissible** to assure that AI could do the best & smart decision.

Depth-Limit HMinMax search algorithm

For the HMinMax algorithm implementation & alpha beta pruning, one more process I add is to clarify if the node to calculate is a terminal node,. If it is a terminal Node, just return the utility value of the node, no more need to do HMinMax search:

```
// the HMinMax plus alpha beta pruning algorithm implementation
private int calHMinMax(AdvanNode node, int a, int b, int depth) {
    if (node.isTerminal()) { // if it is a terminal node, return its utility function
        return node.utility();
    } else if (depth == limitDepth) { // reach the limit depth, use heuristic function
        return node.hValue();
```

This recursive method calHMinMax is to calculate the utility value for a root node(plus the alpha beta pruning). Basic situations are the node is a Terminal Node or it reaches the limit depth(also called cut off test here, the limit depth is a instance variable which is created in the constructor).

The **alpha beta pruning**: as same as the model in Lecture.

```
int value = alphaDefault;
for (int[] action : node.actions()) {
    int board = action[0];// from 1 to 9
    int position = action[1]; // from 1 to 9
    AdvanNode sucNode = node.transition(board, position);
    value = Math.max(value, calHMinMax(sucNode, a, b,  depth: depth + 1));
    if (value >= b) {
        break;
    }
    a = Math.max(a, value);
}
return value;
```

The **action & transition model** here really matters. In the process of search, what we really want to know is which direction the AI should move toward, not just what next node is. The action and transition allow the AI to track both the action and successor node to make de the decision.

The whole process to implement alpha-beta pruning and HMinMax algorithm here is as following:

First, for each successor node of a root node, use calHMinMax calculate it's utility value.

Second, compare all these nodes' utility value, choose a Min value or Max value Node as the next Node AI decide to move forward.

```java
// calculate the solution
private void calSolution() {
    if (root.turn() == GameParameters.firstMove) {
        int value = alphaDefault;
        for (int[] action : root.actions()) {
            AdvanNode node = root.transition(action[0], action[1]);
            int tempValue = calHMinMax(node, alphaDefault, betaDefault,  depth: 1);
            if (tempValue > value) {
                value = tempValue;
                solution = action;
            }
        }
    } else {
        int value = betaDefault;
        for (int[] action : root.actions()) {
            AdvanNode node = root.transition(action[0], action[1]);
            int tempValue = calHMinMax(node, alphaDefault, betaDefault,  depth: 1);
            if (tempValue < value) {
                value = tempValue;
                solution = action;
            }
        }
    }
}
```

(Full code of Decision make in HMinMax search)

A detail here is that what I get from the HMinMax is not what the next node is. The information I use in game process is the action(next move direction), which could be a more clear info for both the player and AI to implement(The project also requires that should print out the move that both AI and Player decide to take).

Another key of the search is how much **depth** we want our AI algorithm to inspect, because this parameter will decide how 'smart' the AI is. This part will be discussed in the last section. The depth here is an instance variable of each HMinMax class.

One critical optimization I have done is that when search the successor nodes of the root node is that when we come across a successor node with utility value equals **victory grade**, should I s**top search immediately or continue search**? After I implement this strategy in both Part 2 and Part 3, it does not performance well.

There are two main reasons I think matter a lot. First, the **depth**. With a shallow depth it is rare to get a terminal Node(whose heuristic value exceeds 1000 or 8000), so this strategy would only performance if set the depth to a lower bound to make sure the search process could reach a terminal Node. Second, the **heuristic value function** I apply to each type of node. In this case, I think a more accurate heuristic function should be used such as if a node **has two three flag lines** and no opponent's three flag line in which case I should **directly give it a victory grade** because after two more rounds the AI must win as in next round the opponent could only constraint one line. These more detailed rules should be applied besides these point based rules.

Game Process

```
protected void initializeState() { state = GameParameters.emptyAdvanNode(size); }

// the Game Process
public void game(int limitDepth) {...}

// endGame method
protected static void endGame(AdvanNode state, int player) {...}

// read board,position that player choose mark
public static int[] readPlayerMove() {...}

// update state in Player's round
protected void updatePlayerRound() {...}

// update state in AI's Round
protected void updateAIRound(int limitDepth) {...}
```

As mentioned in API design part, the method game is a single game, the initializeState method is to offer an initial state to the game(this method will be called each time when a new game started).

```
public void game(int limitDepth) {
    initializeState();
    int player = TTTGame.playerTurn();// use turn input method in TTTGame
    while (!state.isTerminal()) {
        System.err.println(pageLine);
        System.err.println();
        state.showState();
        if (state.turn() == player) {
            System.err.println("It is your round");
            updatePlayerRound();
        } else {
            System.err.println("It is AI's round");
            System.err.println("AI is calculating....");
            updateAIRound(limitDepth);
        }
    }
    // clarify the result, who wins or it is a draw
    state.showState();
    endGame(state, player);
}
```

The code for Game, each time a new state started, firstly read the player's move and initialize the state for the game.

```
public class QubicGame extends Advanced3TGame {
    // constructor
    public QubicGame(int size) { super(size); }

    @Override
    // override method for initial State(for the Qubic Node)
    protected void initializeState() { state = GameParameters.emptyQubicNode(size); }

    public static void main(String[] args) {...}
```

The QubicGame.class(Part 3) just extends the AdvanGame.class(Part 2) and overrides the the initialStatemethod(offer a QubicNode to the Game process):

HMinMax Algorithm analysis:

In the process of implementation of HMinMax algorithm, the alpha beta pruning matters as it make us could search more nodes with same memory usage.

Take Part 2 as example, without the alpha beta pruning, the maximum depth that search could reach is 3 and will cause Java.lang.OutOfMemory exception(Sorry no screen shot here). But with the alpha beta pruning, we could get 9 depth search(could be larger, have not found the max depth could read).

```
public static void main(String[] args) {
    Advanced3TGame game = new Advanced3TGame(GameParameters.classicSize);
    int depth = 9;
    while (true) {
        game.game(depth);
    }
}
```

The main method to run the Part 2 Game process:

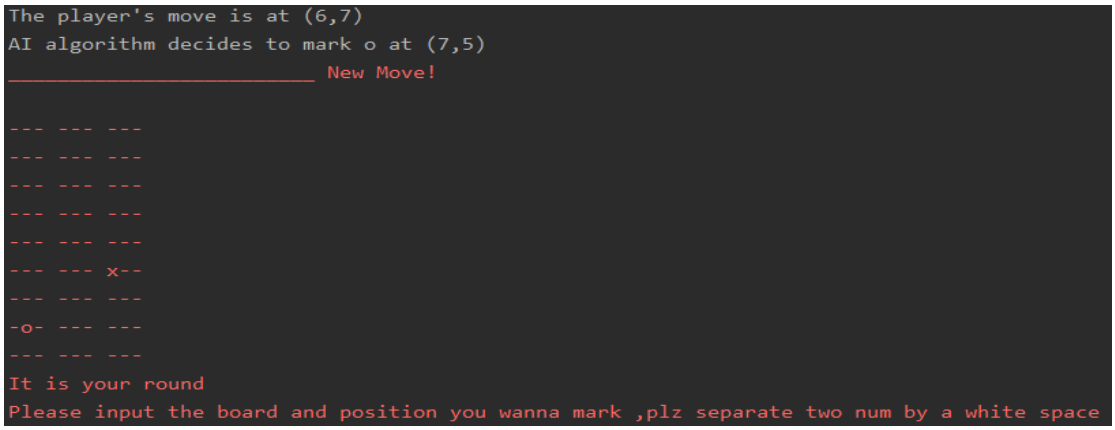
If we choose the x, the AI's result at step 2:

```
AI algorithm decides to mark o at (7,1)
_____________________ New Move!


--- --- ---
--- --- ---
--- --- ---
--- --- ---
--- --- ---
--- --- x--
o-- --- ---
--- --- ---
--- --- ---
It is your round
Please input the board and position you wanna mark ,plz separate two num by a white space
```

Another key of this algorithm is the depth we set to search as it decides how "smart" our AI algorithm is.

For example, if we set the depth to 6 and still mark x at 6,7 at step1, the result that AI give is at (7,5), the fact is that to mark at 7,1 is a better idea for AI algorithm.



However, I have not found the maximum depth that my search program could reach. In theory, it could get 81(in part 2) and 64(in part 3) because it uses depth first search, the space complexity is linear as it always only stores the frontier nodes in memory rather than whole search tree. However, if I set the depth to a large number, the time complexity would increase exponentially. Although there is no time constraint for the project, I do not want my AI go much slower than others.

Stat about the Project

Source File	Total Lines	Source Code Lines	Source Code Lines [%]
Advanced3TGame.java	98	80	82%
AdvanNode.java	202	159	79%
GameParameters.java	132	73	55%
HeuristicAdvanNode.java	79	61	77%
HeuristicQubicNode.java	39	34	87%
HMinMax.java	91	76	84%
QubicGame.java	22	16	73%
QubicNode.java	298	235	79%
TTTGame.java	159	133	84%
TTTNode.java	183	144	79%
Total:	**1303**	**1011**	**78%**