Tic-Tac-Toe Project Report

Wei Zhang

1. Introduction of rules

There are three versions of Tic-Tac-Toe game in our project, basic version, advanced version and ultimate version. The algorithms we use for the computer player are all variants of MINIMAX.

The process of basic Tic-Tac-Toe is that two players, X and O, who take turns marking the spaces in a 3×3 grid. The player who succeeds in placing three of their marks in a horizontal, vertical, or diagonal row wins the game. [1]

Advanced version expands the board to a 9*9 one, which is divided by a 3*3 grid. The players still take turns to mark the board, but the mark has to be in the grid that corresponds to the position in the board where the previous player marked. (For example, if a player marks the bottom right position, of any board, then the next player must mark any open space on the bottom right board.) But there are two exceptions when the player can mark in any grid: 1. At the beginning of the game. 2. When the board in which the player should mark is full. Each grid is considered as an individual board. If a player wins in a grid, the player wins in the whole game.

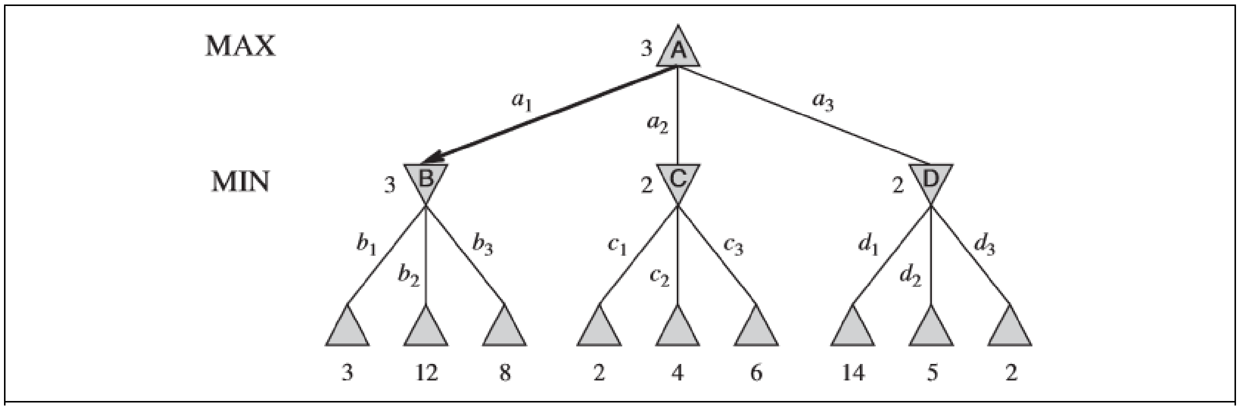
Ultimate Tic-Tac-Toe are mostly similar to the advance version, but a player has to win 3 grids in row horizontally, vertically, or diagonally instead of just one. Besides, this version has one more exception in which the next player can randomly choose the grid: a board is not full but has already got a winner.

2. Algorithms Applied

For basic Tic-Tac-Toe, we use simple MINIMAX algorithm [2], which explores all the successors of the current state to get the utilities for different actions.

```
function MINIMAX-DECISION(state) returns an action
  return arg max_{a ∈ ACTIONS(s)} MIN-VALUE(RESULT(state, a))
-------------------------------------------------------------
function MAX-VALUE(state) returns a utility value
  if TERMINAL-TEST(state) then return UTILITY(state)
  $v \leftarrow -\infty$
  for each a in ACTIONS(state) do
    $v \leftarrow$ MAX(v, MIN-VALUE(RESULT(s, a)))
  return v
-------------------------------------------------------------
function MIN-VALUE(state) returns a utility value
  if TERMINAL-TEST(state) then return UTILITY(state)
  $v \leftarrow \infty$
  for each a in ACTIONS(state) do
    $v \leftarrow$ MIN(v, MAX-VALUE(RESULT(s, a)))
  return v
```

In this algorithm, we do a depth-first search to recursively explore the entire tree below the current state. When it finds a terminal state, the utility function will return a value according to which player wins. For example, If the state indicates that the player who take X wins, then the utility for X is 1 and -1 for O and vice versa. Especially, when the players get tie, the utilities for them are both 0. When the MINIMAX algorithm is looking for the optimal solution as a player, it assumes that its opponent will take the best choice for every step as well, while the best choice for one player is the worst for the opponent in the zero-sum game. So, when it is the computer's turn to move, it will take the action that maximize the utility for computer. When it is the opponent's turn, it will take the action that minimize the utility for the computer. In this way, the computer can finally get all the utility for itself of all the succeeding states and make the best choice according to that.



For advanced Tic-Tac-Toe, we use the H-MINIMAX algorithm [2] with alpha-beta pruning [2]. In this part, we limit the depth of the MINIMAX search to a fixed value. If we cannot find a terminal state until that fixed depth, we will use the heuristic function to evaluate the state and use it as the utility.

$$\text{H-MINIMAX}(s,d) = \begin{cases} \text{EVAL}(s) & \text{if CUTOFF-TEST}(s,d) \\ \max_{a \in Actions(s)} \text{H-MINIMAX}(\text{RESULT}(s,a), d+1) & \text{if PLAYER}(s) = \text{MAX} \\ \min_{a \in Actions(s)} \text{H-MINIMAX}(\text{RESULT}(s,a), d+1) & \text{if PLAYER}(s) = \text{MIN}. \end{cases}$$

The heuristic function evaluates the state as following: (with respect to X)

In every grid of the board:

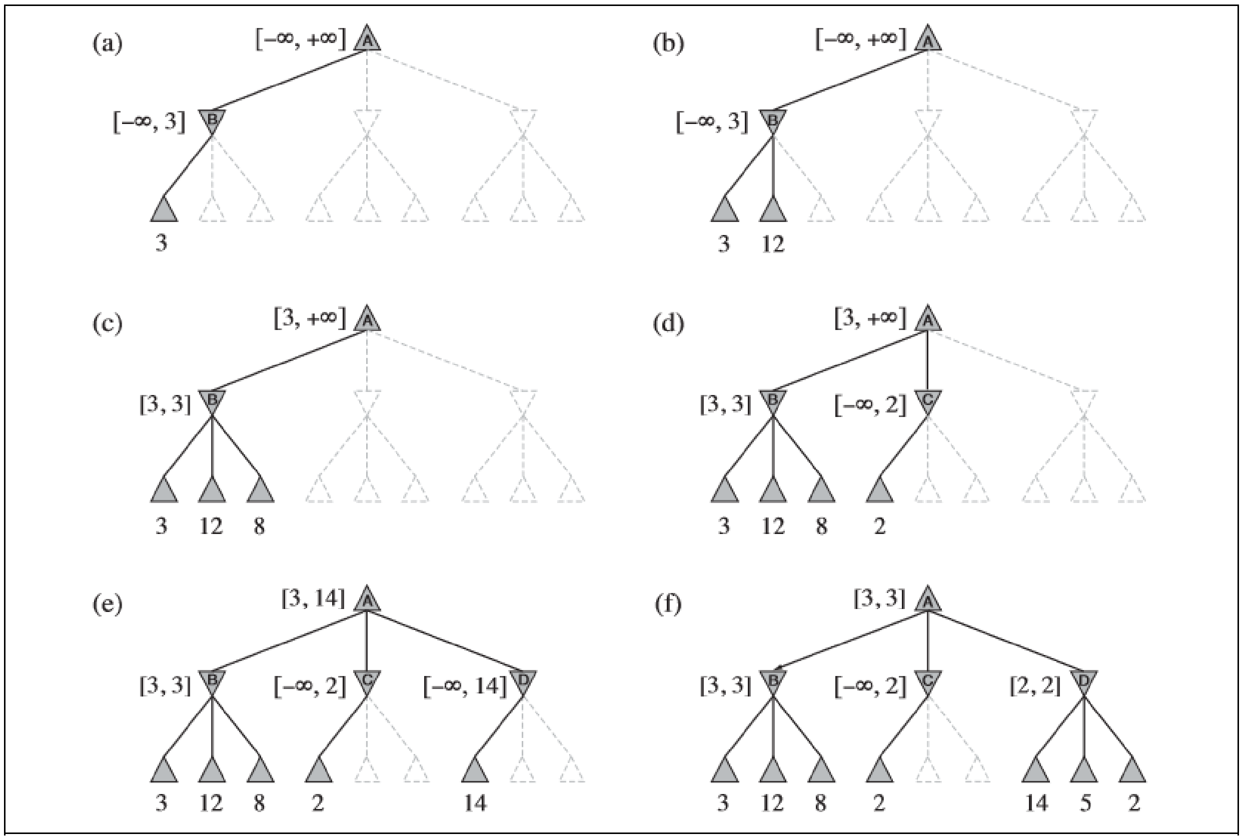
1. If there are two X marks in a vertical, horizontal or diagonal row, the evaluation value of this grid will increase by 1.

2. If there are three X marks in a row, which means X wins, the evaluation value of this grid will increase by 100.

3. If there are more X marks in the grid than O, then the evaluation value will increase by 5 times the difference between the number of X and O marks.

The heuristic function will also be applied to O, and the final evaluation value will be the difference between the evaluation values of these two values. The evaluation value of the state is the sum of the final values of all the grids.

In order to save time, we apply alpha-beta pruning algorithm during the search. Suppose that we are searching the tree below, which has two layers as MAX and MIN from top to bottom. After exploring the



subtree with state B as the root, we find that from the subtree we will get no more than 3 as the evaluation value. So, when we begin exploring subtree C and we find a state with an evaluation value, 2, which indicates that the subtree C will get an evaluation value no more than 2. Then we can stop since that subtree can not affect the value of A anymore. This is how alpha-beta pruning saves the searching time.

For the ultimate Tic-Tac-Toe, we only change the heuristic function based on the advanced one.

For three grids in a horizontal, vertical, or diagonal row:

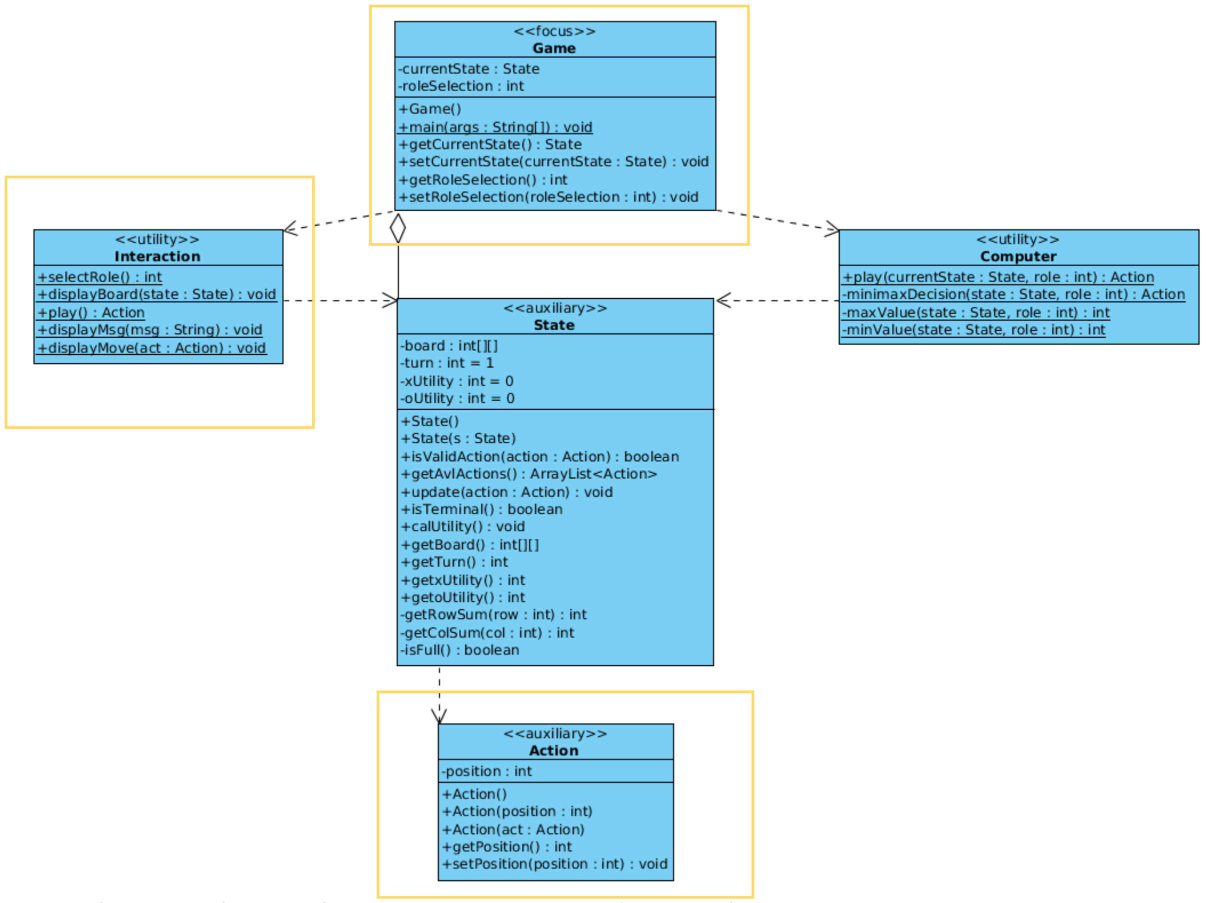
1. If the evaluation values of two grids for X in a row is greater than those for O, the evaluation value of this state will increase by 10.

2. If the evaluation values of three grids for X in a row is greater than those for O, the evaluation value of this state will increase by 100.

The heuristic function will also be applied to O, and the final evaluation value will be the difference between the evaluation values of these two values.

3. System Structure (My Work)

This game is designed in a Model-View-Controller (MVC) pattern. Model layer contains two classes, Action and State (and one more class, Board, for advanced TTT and ultimate TTT), which are the data structures used in this system. Controller layer contains two classes, Computer and Game, which are responsible for the actions the system takes. View layer contains only one class, Interaction, which interacts with the user by receiving inputs and print messages to the standard streams.


<<focus>>
Game
-currentState : State
-roleSelection : int
+Game()
+main(args : String[]) : void
+getCurrentState() : State
+setCurrentState(currentState : State) : void
+getRoleSelection() : int
+setRoleSelection(roleSelection : int) : void
<<utility>>
Interaction
+selectRole() : int
+displayBoard(state : State) : void
+play() : Action
+displayMsg(msg : String) : void
+displayMove(act : Action) : void
<<utility>>
Computer
+play(currentState : State, role : int) : Action
-minimaxDecision(state : State, role : int) : Action
-maxValue(state : State, role : int) : int
-minValue(state : State, role : int) : int
<<auxiliary>>
State
-board : int[][]
-turn : int = 1
-xUtility : int = 0
-oUtility : int = 0
+State()
+State(s : State)
+isValidAction(action : Action) : boolean
+getAvlActions() : ArrayList<Action>
+update(action : Action) : void
+isTerminal() : boolean
+calUtility() : void
+getBoard() : int[][]
+getTurn() : int
+getxUtility() : int
+getoUtility() : int
-getRowSum(row : int) : int
-getColSum(col : int) : int
-isFull() : boolean
<<auxiliary>>
Action
-position : int
+Action()
+Action(position : int)
+Action(act : Action)
+getPosition() : int
+setPosition(position : int) : void

My work is to implement the Interaction, Game and Action class.

The Interaction class has five functions:

selectRole(): Prompt the user to select the role and return the choice.

displayBoard(State): Print the current board to standard error.

play(): Prompt the user to input the next move and return it as a class Action.

displayMsg(String): Display messages to standard error.

displayMove(Action): Display the program’s move to standard output.

The Game class contains the main function of this system, it starts the game and manage the activities in the game until it finishes. Then it announces the winner and restarts the game. Meanwhile, this class needs to ask the player for the next action and validate the action. If the action is invalid, it should keep asking for another action until it is valid.

The Game class has 6 functions, but only 2 are worth to introduce:

Game(): initialize the game. First, initialize the board, and ask for the player which side to take.

Main(): Iteratively start the game. Ask for actions. Validate actions and apply actions. Iterate until a terminal state. Announce the winner.

The Action class keeps the information of all the actions used in the game. It varies in different version of this system. It only contains some getter and setter functions and several constructor functions including a copy constructor.

4. Result and Analysis

The basic TTT works ultimately. Nobody can defeat it (the best result you can get is tie). We fix all the problems we encountered, including invalid inputs like 'ab dc' and '1   3'. The time consumption is pretty little.

The advanced TTT works pretty well. Most time we cannot defeat it. But when we are competing with another group's program, our program can win only if we act after theirs. The time consumption of this program is also little if we limit the max search depth to 9 or less.

The ultimate TTT works pretty well as well. It seems that it is trying to bring the opponent to a specific grid so that it can get advantage in other grids. It is very clever to do so. The time consumption of this program is also little if the max search depth is lower than 8.

Actually, I think there is a balance between the time consumption and the performance.

Collaborator:

Hao Huang (NetID: hhuang40)

Announcement:

All the pictures of algorithm is from [2].

References:

[1] https://en.wikipedia.org/wiki/Tic-tac-toe

[2] Artificial Intelligence - A Modern Approach, 3rd Edition