Project 1

Tic-Tac-Toe

Hao Huang (NetID: hhuang40)

February 13, 2018

1. Definition of Class

In this section, we will present all classes we use in this project. Although we finish three versions of Tic-Tac-Toa, for the sake of code reuse, the classes in each version are almost the same (except for core algorithms).

1.1 Basic TTT

1. State
   For the basic version, the *State* class has a member attribute: *board*, a 3×3 int array, where each entry represents a location on the board. Initially, all entries are set to 0, indicating there is not any chess piece (X or O) on the board. If an X is placed on a certain position, its value will change to $+1$. If an O is placed, the value will be set to -1. The other member attributes and functions are listed in Figure 1.

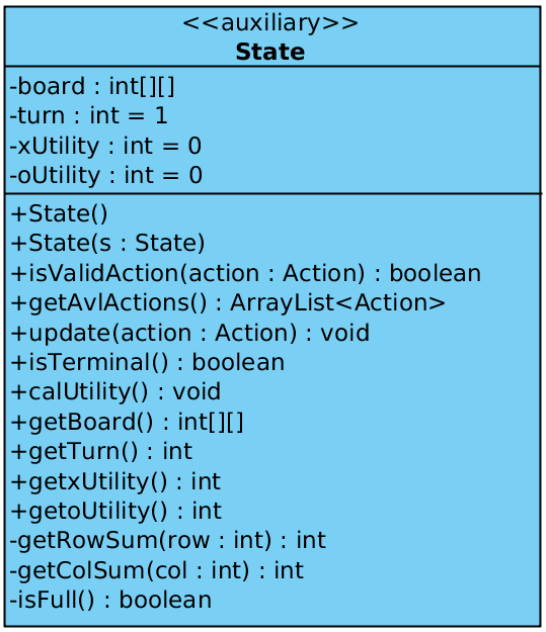

Figure 1: State class used in basic Tic-Tac-Toe

Member attribute *turn* records which player (X or O) is now playing. *xUtility* and *oUtility* are used to store a numeric utility. Here we adopt two utilities for the ease of calculation and parameter passing. When a game ends with a tie, they are equal. Otherwise, they are additive inverse to each other.

Except for getters and setters, *isValidAction()* aims at checking whether a given action is applicable at current state, and *getAvlActions()* will generate all available actions at current state during tree search process. The function *update()* applies an action to current state, which is somewhat like "RESULT()" in AIMA (page 162). The

function *calUtility* will compute *xUtility* and *oUtility*. The function *isFull* is to check whether the board is full.

2. Action
   This class represent the action a player takes, as shown in Figure2. It is implemented by my partner, more details can be found in his report.

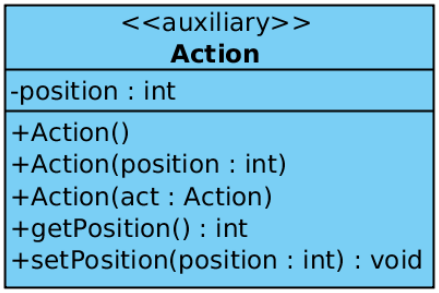

Figure 2: Action class used in basic Tic-Tac-Toe

3. Computer
   The *Computer* is a static class and contains the core Algorithm *MINIMAX*, acting as a computer player. Apart from the function *play()* which is an entry-function, other functions are translated from the pseudocode from AIMA (page 166). We will talk about it in detail in Section 3.1. This class is shown in Figure3. Note that we add one more parameter "*role*" in each member function, aiming at discriminate who is player now.

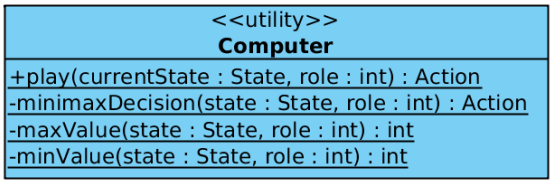

Figure 3: Game class used in basic Tic-Tac-Toe

4. Interaction
   This class can be seen as a counterpart to *Computer* class. It deals with all matters involving with human interaction, such as reading in human player's input, displaying board, printing (result or error) messages, etc. *selectRole()* is used to ask the human player to select X or O when a new game begins. *play()* is the entry function which will be called every step when it is human's turn. More information will be provided in my partner's report.

5. Game
   The *Game* class is a control module, which initializes an empty state when a new game begins, as shown in Figure4. The member attribute *currentState* is an instantiation of *State* and represents the board. *roleSelection* records whether human or computer acts as X. The main function in this class will call *Computer.play()* and *Interaction.play()* alternatively to drive the game forward. To check whether the game is over, this class will call *currentState.isTerminal()*. To judge the result of a game, *currentState.calUtility()* will be called and X's and O's utilities will be compared. More information will be provided in my partner's report.

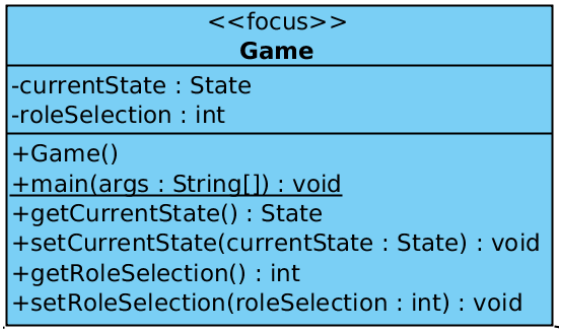

Figure 4: Computer class used in basic Tic-Tac-Toe

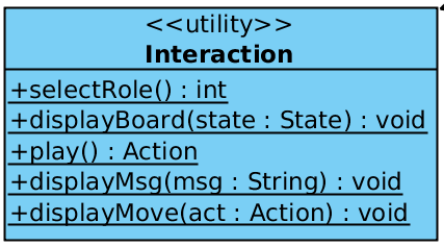

Figure 5: Interaction class used in basic Tic-Tac-Toe

1.2 Advanced TTT

The advanced version of Tic-Tac-Toe contains a big grid which consists of nine board arranged in 3×3. Realizing this fact, we incorporate a new class *Board* and modify the state *State*.

1. Board
   As we can see from Figure6, the *Board* class is very similar to the *State* class in the basic version, and the primary member attribute is a 3×3 int array representing a board. *xEvaluation* and *oEvaluation* is analogy to *xUtility* and *oUtility* in the basice

version. An additional member attribute is *grididx*, ranging from 1 to 9. It records the position this board occupies on a big grid. *calEvaluation()* is used to compute numeric value of heuristic function, which will be discussed in Section 4. Other member functions are self-explanatory.

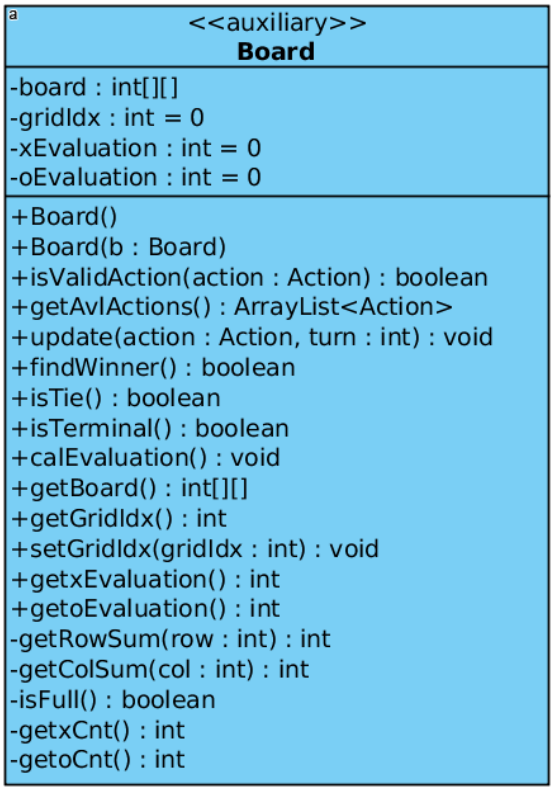

Figure 6: Board class used in advanced Tic-Tac-Toe

2. State
   State class in advanced version contains a grid which is a 3×3 *Board* array. *maxDepth* records current search depth which will be used in depth-limited search. *lastAction* keeps track of each action applied to current state. It can be used in checking the applicability of actions. Note that this class also contains *xEvaluation* and *oEvaluation*. These two values are derived from *Board.xEvaluation* and *Board.oEvaluation*, which will be discussed in Section 4. The class is shown in Figure7.

   The only one member function we need to mention here is *isCutoff()*. When a game terminates (tie or having a winner), or the searching depth reaches the pre-defined maximum depth, *isCufoff()* will return true and search will stop.

3. Computer
   The *Computer* is a static class and contains the core Algorithm *H-MINIMAX* with

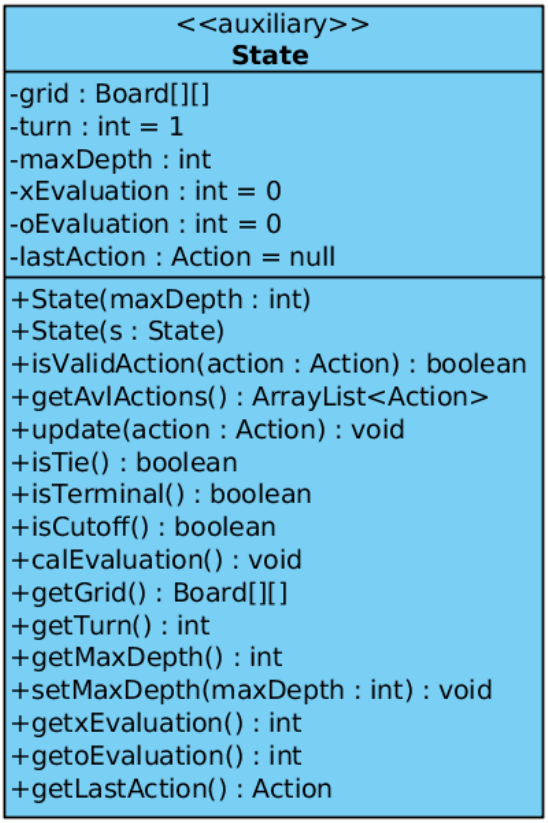

Figure 7: State class used in advanced Tic-Tac-Toe

$\alpha - \beta$ pruning, acting as a computer player. Like the cognominal class in the basic version, this class realizes pseudocode from AIMA (page 170). We will talk about it in detail in Section 3.2. All member functions in class are shown in Figure8.

1.3 More Advanced TTT

All classes adopted in the more advanced version is the same as advanced version. Therefore we will not explain them once again. The only difference is that the more advanced version uses more complex heuristic function and termination criterion.

2. Architecture

All versions are developed using object-oriented programming. In this section we present the UML we design during developing.

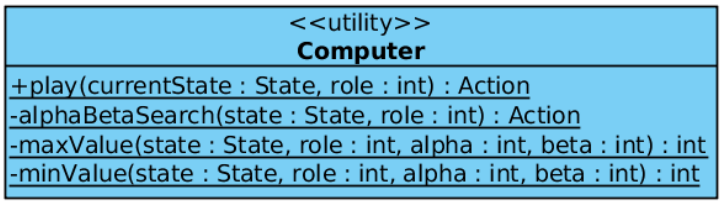

Figure 8: Computer class used in advanced Tic-Tac-Toe

2.1 Basic TTT

As shown in Figure9, since *Game* contains an instantiation of *State*, the relation between them is aggregation. Other relations are dependency. One class might pass another class as a parameter, or return anther class.

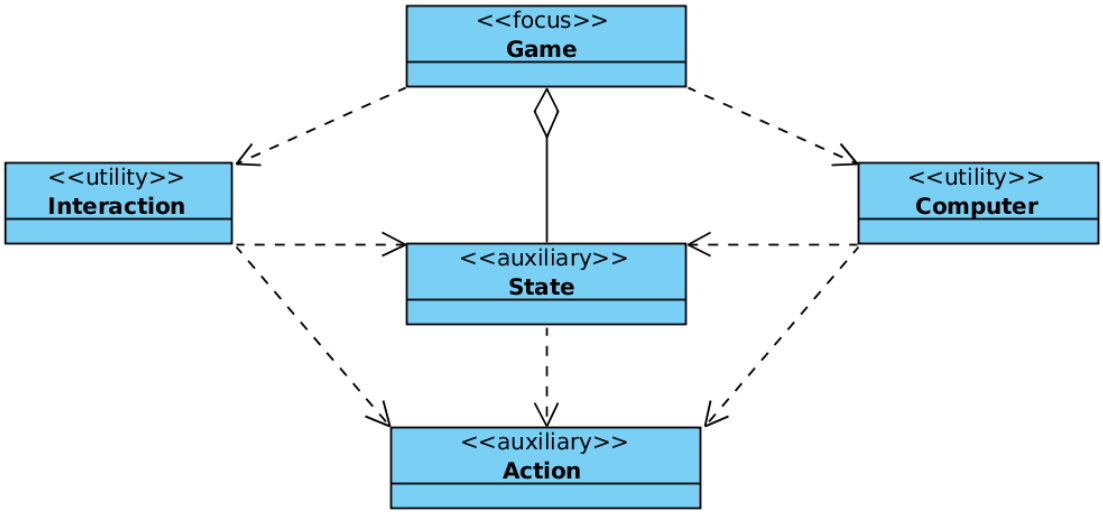

Figure 9: UML for basic Tic-Tac-Toe

2.2 Advanced TTT and More Advanced TTT

The UML (Figure10) of the advanced version is similar to that of the basic version. The main difference is the relation between *State* and *Board* is aggregation, since *State* contains an array of instantiations of *Board*. The architecture of the more advanced version is the same as the advanced one, so we omit it here for conciseness.

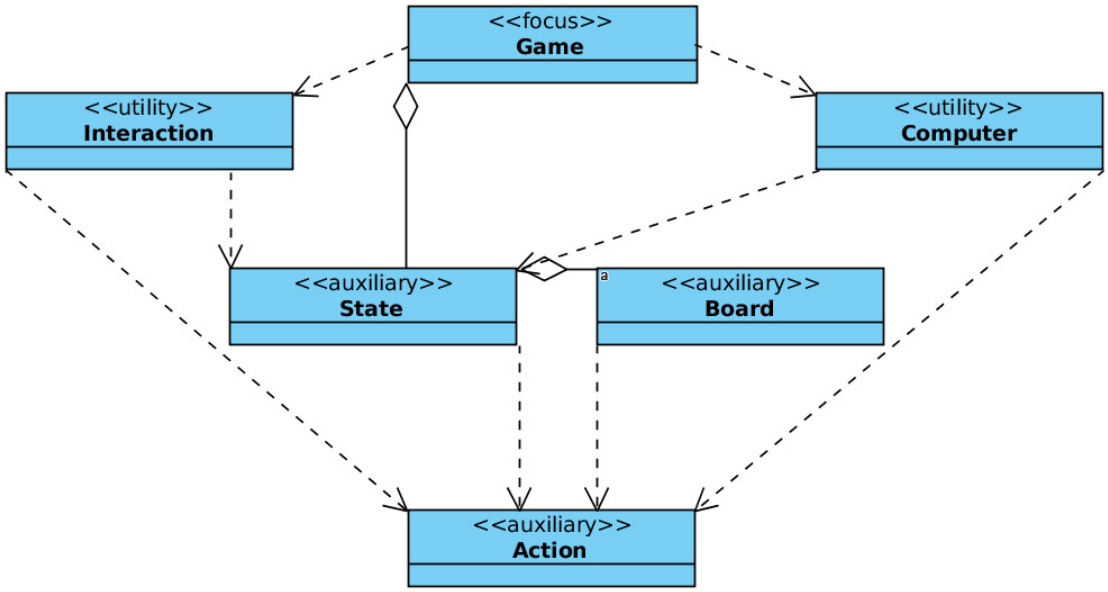

Figure 10: UML for (more) advanced Tic-Tac-Toe

3. Core Algorithm

In the basic version of Tic-Tac-Toe, we adopt the algorithm of *MINIMAX*.In the advanced and more advanced version, we adopt the algorithm of *HMINMAX* with $\alpha - \beta$ pruning.

3.1 Minimax Algorithm

The pseudocode of Minimax is show in Figture11[1]. In this example, the state is a board with 3×3 locations. Applicable actions are placing a chess piece (X or O) at an empty location which is represented by 0 on the board. We use a loop to find all applicable actions. If a X is place at a certain location, its value will change to 1. Otherwise, the location value will change to -1. The utility is calculated based on terminal states. If X wins a game, *xUtility* will set to $+1$. If X loses, *xUtility* will set to -1. The same criterion for O. If a game ends with a tie, both *xUtility* and *oUtility* will set to 0. Based on we've discussed above, it is not a hard job to turn this pseudocode into Java code.

3.2 HMinimax Algorithm

The pseudocode of H-Minimax with $\alpha - \beta$ pruning is show in Figture12[1]. Advanced and more advanced version adopt this algorithm. To make it more concrete, the state is a grid with 3×3 boards and each board has 3×3 int locations. Each action is represented by a tuple of two integers $< i, j >$, with the first one referring to the board location on the grid, and the second one referring to the integer location on the board. In addition to the rule in the basic version, an applicable action must satisfy $j_{old} = i_{new}$. (Another rule for more

```
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 ← −∞
  for each a in ACTIONS(state) do
    v ← 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 ← ∞
  for each a in ACTIONS(state) do
    v ← MIN(v, MAX-VALUE(RESULT(s, a)))
  return v
```

Figure 11: Pseudocode for Minimax algorithm

advanced version is: player cannot place a chess piece on any terminal board, say, a tie or having a winner.) We use two loops to find all applicable actions. The outer loop for grid and the inner loop for board.

1. Limited-depth Search
   To avoid long time search in a very deep state-space tree, we set the maximum search depth. Therefore, the TERMINAL-TEST() in this pseudocode will be replaced with CUTOFF-TEST(). When the maximum depth is reached or a winner (or a tie) appears, the cutoff test will return true. In this project, we set the maximum depth to 7 for advanced version, and 7 for more advanced version by default. Such values are tradeoff between winning probability and running time.

2. $\alpha - \beta$ Pruning
   α records the best value we have found so far from MAX viewpoint, and β records the best value we have found so far from MIN viewpoint. This technique aims at improve search efficiency by pruning all branches (and leaves) which cannot be chosen as the best action. α and β are just passing as parameters as the pseudocode, not reflected in data structure.

4. Heuristic Function

In *HMINIMAX* algorithm, UTILITY() should be replaced with EVALUATION(). However, for more advanced version, we adopt a more complex version of heuristic function, since a

```
function ALPHA-BETA-SEARCH(state) returns an action
  v ← MAX-VALUE(state, −∞, +∞)
  return the action in ACTIONS(state) with value v

function MAX-VALUE(state, α, β) returns a utility value
  if TERMINAL-TEST(state) then return UTILITY(state)
  v ← −∞
  for each a in ACTIONS(state) do
    v ← MAX(v, MIN-VALUE(RESULT(s,a), α, β))
    if v ≥ β then return v
    α ← MAX(α, v)
  return v

function MIN-VALUE(state, α, β) returns a utility value
  if TERMINAL-TEST(state) then return UTILITY(state)
  v ← +∞
  for each a in ACTIONS(state) do
    v ← MIN(v, MAX-VALUE(RESULT(s,a) , α, β))
    if v ≤ α then return v
    β ← MIN(β, v)
  return v
```

Figure 12: Pseudocode for H-Minimax algorithm with $\alpha - \beta$ pruning

player must win three boards in a grid horizontally or vertically or diagonally.

4.1 Advanced TTT

For the sake of simplicity, we take the role of X as an example. We define two criteria to evaluation a state of a board (Note: not a grid).

1. The number of X on a row/column/diagonal without O
   If there are three Xs on a row/column/diagonal, we assign *xEvaluation* a large positive number, 100. If there are two Xs on a row/column/diagonal, and the left location on the same row/column/diagonal is empty, we assign *xEvaluation* to a small positive number 1. If *xEvaluation* is positive, *oEvaluation* will be negative, and vice versa.

2. The difference of the number of X and the number of O
   We use $\#X - \#O$ as the basis. More concretely, if there are 6 X pieces and 3 O pieces on a board, *xEvaluation* will add $(6-3) \times 5 = 15$. One more X, more 5 points will be added to *xEvaluation*. If *xEvaluation* is positive, *oEvaluation* will be negative, and vice versa. It has been mentioned above.

The first criterion forces computer to choose actions which are likely to lead to success,

and the second criterion forces computer to pay more attention to the board on which there are too many human's chess pieces. We add the values derived from the two above criteria together as the evaluation of a single board.

If a player wants to win a game, he only needs to win any single board on a grid. Therefore, the numeric evaluation value of a state of a grid (Note: not a board) is the sum of all evaluation values of nine boards. This design is relative simple, but rather effective.

4.2 More Advanced TTT

The heuristic function of a single board (criterion 1) in this version is the same as that of the advanced version. Criterion 2 becomes more complex since it is hard for any player to win the game.

Here we apply the criterion 1 which is applicable to a board again to a grid. More concretely, if X wins three boards horizontally or vertically or diagonally, we add 100 to *xEvaluation*. If X wins two boards horizontally or vertically or diagonally, and the other board on the same row/column/diagonal remains unfinished or is a tie, we add 10 to *xEvaluation*. No positive gain will be assigned to *xEvaluation* for any other grid situation. A positive heuristic value for *xEvaluation* means a negative heuristic value for *oEvaluation*, and vice versa.

5. Demo Snippet

Here we present several snippets of the running effects of the program. This is only for a very simple illustration. Running time for each time of search has already been printed to System.err. During our experiment, the more maximum search depth, the longer time the program will take.

5.1 Basic TTT

As shown in Figure13, "5" is human input and "1" is the outcome produced by computer (or algorithm).

5.2 Advanced TTT

As shown in Figure14 and Figure15, "5 5" is human input and "5 1" is the outcome produced by computer (or algorithm).

References

[1] Russell, S. J., Norvig, P., Canny, J. F., Malik, J. M., & Edwards, D. D. (2003). *Artificial intelligence: a modern approach* (Vol. 2, No. 9). Upper Saddle River: Prentice hall.

```
Welcome to Tic-Tac-Toe
Please select your side (  X o
x
The current board status is:
-------
| | | |
-------
| | | |
-------
| | | |
-------

Please select your next move:
5
5
The current board status is:
-------
| | | |
-------
| |X| |
-------
| | | |
-------

1
The current board status is:
-------
|O| | |
-------
| |X| |
-------
| | | |
-------
```

Figure 13: Snippet for basic Tic-Tac-Toe

```
Welcome to Tic-Tac-Toe
Please select your side (  X or O ):
x
The current board status is:
-----------------------
|---------------------|
||  |  |  ||  |  |  ||  |  |  ||
||  |  |  ||  |  |  ||  |  |  ||
||  |  |  ||  |  |  ||  |  |  ||
|---------------------|
||  |  |  ||  |  |  ||  |  |  ||
||  |  |  ||  |  |  ||  |  |  ||
||  |  |  ||  |  |  ||  |  |  ||
|---------------------|
||  |  |  ||  |  |  ||  |  |  ||
||  |  |  ||  |  |  ||  |  |  ||
||  |  |  ||  |  |  ||  |  |  ||
|---------------------|
-----------------------
Please select your next move:
5 5
5 5
The current board status is:
-----------------------
|---------------------|
||  |  |  ||  |  |  ||  |  |  ||
||  |  |  ||  |  |  ||  |  |  ||
||  |  |  ||  |  |  ||  |  |  ||
|---------------------|
||  |  |  ||  |  |  ||  |  |  ||
||  |  |  ||  |X|  ||  |  |  ||
||  |  |  ||  |  |  ||  |  |  ||
|---------------------|
||  |  |  ||  |  |  ||  |  |  ||
||  |  |  ||  |  |  ||  |  |  ||
||  |  |  ||  |  |  ||  |  |  ||
|---------------------|
-----------------------
```

Figure 14: Snippet for advanced (more advanced) Tic-Tac-Toe

```
5 1
The current board status is:
----------------------
|--------------------|
||  |  |  ||  |  |  ||  |  |  ||
||  |  |  ||  |  |  ||  |  |  ||
||  |  |  ||  |  |  ||  |  |  ||
|--------------------|
||  |  |  ||O|  |  ||  |  |  ||
||  |  |  ||  |X|  ||  |  |  ||
||  |  |  ||  |  |  ||  |  |  ||
|--------------------|
||  |  |  ||  |  |  ||  |  |  ||
||  |  |  ||  |  |  ||  |  |  ||
||  |  |  ||  |  |  ||  |  |  ||
|--------------------|
----------------------
Please select your next move:
```

Figure 15: Snippet for advanced (more advanced) Tic-Tac-Toe (cont.)