CSC442: Introduction to AI

Project 1: Game Playing

Haoyuan Yu

Part I: Basic Tic-Tac-Toe

1. Formulation

The tic-tac-toe is a game for two players, who take turns to place their own marks, 'x' and 'o', in a 3*3 grid. The rule of the game is easy: the player who first placing three of their marks in a horizontal, vertical, or diagonal row wins the game. Obviously, this game is a sum-zero game, which means that there are only three results of the game: player I wins, and player II loses; player II wins, and player I loses; draw game.

In this project, our goal is to design an AI agent who can play the game with other players automatically and ensure to perform well. In other words, the AI agent must maximize its chance to win in a given situation.

Let's first define the problem. The tic-tac-toe can be formally defined with the following elements:
(1) The initial state S_0: In order to make the state easy to access and rewrite, I decided to use dictionary in python as the data structure to store all the information in a given state. The dictionary contains four key-value pairs. Here is an example of the state, which is also the initial state of the problem:

```
{ 'player' : ([ ],'x'), 'comp' : ([ ],'o'), 'left' : [1,2,3,4,5,6,7,8,9], 'turn' : -1 }
```

In the initial state, neither the player nor the computer (AI) have taken any moves yet, so the list corresponding to 'player' and 'comp' are empty, and the list corresponding to 'left', containing all the possible moves so far from 1 to 9. In this situation, the player chose 'x' so the mark has been stored in the value of the kay 'player', so does the mark of the computer. Besides, the key 'turn' indicates who is going to take a move next. If the value is -1, then the player should move next and if the value is 1, then the computer should move next.

(2) Actions. In the tic-tac-toe game, the "Actions" is the set of moves that can be take at a given state. Specifically, it's the set of the positions that left unfilled by the player and the computer. In our way to define the state, we can check the list corresponding to 'left' to get all the legal actions. In the initial state, actions are [1,2,3,4,5,6,7,8,9].

(3) Result(s,a). A "Result" function will return a new state by taking an action on the

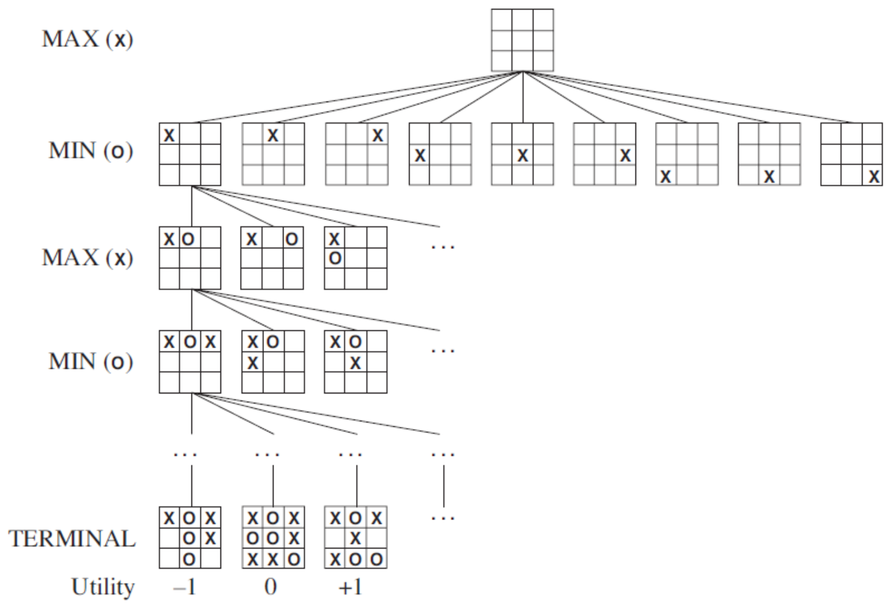
previous state. In my method, I just remove the corresponding number from 'left' and add it to 'comp' or 'player'. Then I multiply the 'turn' by -1. For example, if the player takes the move 5 given the initial state above, then the "Result" will return:

{ 'player' : ([5],'x'), 'comp' : ([ ],'o'), 'left' : [1,2,3,4,6,7,8,9], 'turn' : 1 }

(4) Terminal-Test(s). The "Terminal-Test" will return whether the game is over. Three situations will end the game: 1), the player has a set of moves that include three consecutive numbers in a horizontal, vertical, or diagonal row and it wins; 2), the computer has this set of moves and it wins; 3) all the positions have been filled and no one get to a win, then there is a tie game.

(5) Utility(s). The terminal-test will evaluate the given state and use utility function to give it a score. For this game, the utility function is easy. If the computer wins, the score is 1; if the player wins, the score is -1; otherwise, the score is 0.

Now we have the elements above so that we can define the game tree for the game. In this tree, each node represents a state, and the task of AI agent is to do a depth-first search on all the possible states and find the one with the most utility score.



2. Algorithms

The tic-tac-toe, unlike other problems to be solved by AI, has a more-than-one -agent environment, or a multiagent environment, in which each agent must consider not only their own actions, but also other agents' actions and effects. Therefore, the AI agent need to repeatedly evaluate the situation and all possible actions based on the current state every time after the other agent take an action, which is called the adversarial search.

The Algorithm suitable to do the adversarial search is the Minimax Algorithm. Based on my utility function, I'll let the computer (AI) to play the "Max" role and let the player to play the "Min" role. We assume that the computer will always choose the action that will lead to a state with maximal value and the player will always choose the minimal one. Based on that, the minimax algorithm will use a recursive computation of the minimax values of each successor state to give an action for the "Max" and "Min" players. Here is the pseudocode for this algorithm.

```
function MINIMAX-DECISION(state) returns an action
  return $\arg\max_{a \in \text{ACTIONS}(s)}$ 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
```

3. Code Implementation

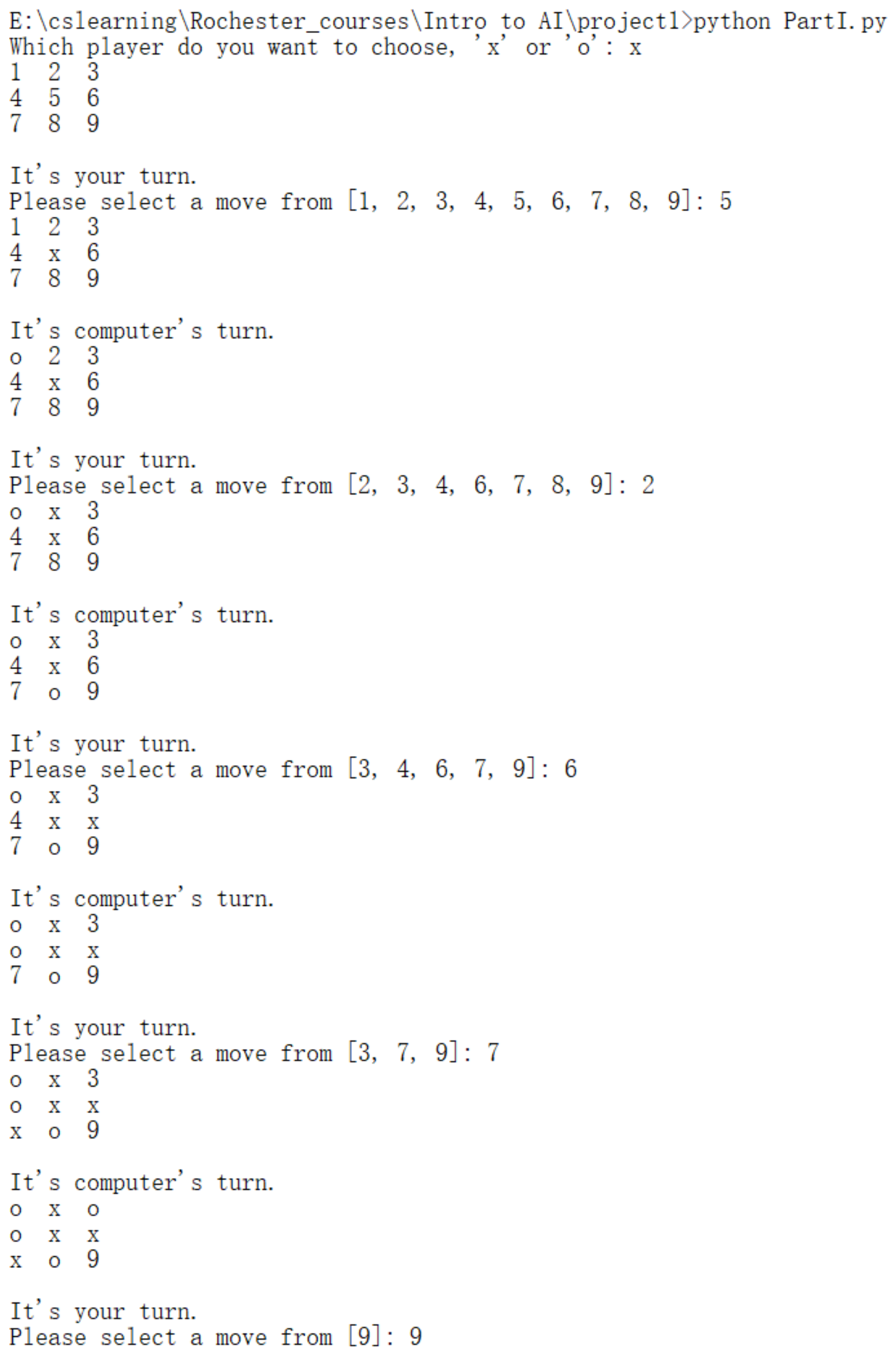
Since the tic-tac-toe game is not complicated and the state is easy to define, I only create one class named tic-tac-toe.

The most important function is implement(). In this function, I designed a work flow for the whole program:

1) I first call the initialize() function to ask the player to choose a side and then create an initial state.
2) After that, the program entered a loop. In each iteration, the program will check whose turn it is. If it's computer's turn to make a move, then the action_search(s) function will be called to tell the computer to search for an action.
   i. The action_search(s) function will first call the check_state(s) function to check whether the game is over. If yes, it will return None.
   ii. Otherwise, it recursively calls the Min_value(s) function and the Max_value(s) function to find the best action.
3) If it's the player's turn to make a move, then the action_ask(s) function will be called to ask the player to select a move.
   i. The action_search(s) function will first call the check_state(s) function to check whether the game is over. If yes, it will return None.
   ii. Otherwise, it ask the player to make a move and return that move.
4) The program will check the return value of action_search(s) of action_ask(s) function.

i. If it’s None, then it means the game is over and it will break the loop and call the print_result(s) function and end the program.
ii. Otherwise, it will call the change_state(s, a) function to create a new state and enter the next iteration.

4. Demo



```
E:\cslearning\Rochester_courses\Intro to AI\project1>python PartI.py
Which player do you want to choose, 'x' or 'o': x
1  2  3
4  5  6
7  8  9

It's your turn.
Please select a move from [1, 2, 3, 4, 5, 6, 7, 8, 9]: 5
1  2  3
4  x  6
7  8  9

It's computer's turn.
o  2  3
4  x  6
7  8  9

It's your turn.
Please select a move from [2, 3, 4, 6, 7, 8, 9]: 2
o  x  3
4  x  6
7  8  9

It's computer's turn.
o  x  3
4  x  6
7  o  9

It's your turn.
Please select a move from [3, 4, 6, 7, 9]: 6
o  x  3
4  x  x
7  o  9

It's computer's turn.
o  x  3
o  x  x
7  o  9

It's your turn.
Please select a move from [3, 7, 9]: 7
o  x  3
o  x  x
x  o  9

It's computer's turn.
o  x  o
o  x  x
x  o  9

It's your turn.
Please select a move from [9]: 9
```

Part II: Advanced Tic-Tac-Toe

1. Formulation

The role of this game is similar with that of the basic tic-tac-toe. However, in this game the two players will play on a 3x3 tic-tac-toe boards arranged in a 3x3 grid. Except for the first move, the two players have to play on the board that share the same number with the last move of their opponent. But if the corresponding board is full, they can play on any board.

According to the new rule, some changes need to be made on the definition of the problem:

(1) The initial State S_0. We redefine the state generally by creating nine dictionaries that similar with that in Part I. Here is the one of the initial states:

    [{'player' : [ ], 'comp' : [ ], 'left' : [1,2,3,4,5,6,7,8,9] } for _ in range(9)]

    Besides, I added some other attributes to the initial state: “turn” indicates whose turn it is to make the next move; “order” records who choose to play first in the game; “last_move” records the last move of a given state and its value is None in a initial state; “nonfull” is a set storing sequence numbers of boards that are not yet be completely filled.

(2) Actions are substantially the same with that in Part I. When the board corresponding to the last move is not full, then the action will be taken on that board, while when the board is full, then the action can be taken on any non-full boards.

(3) Result(s,a) will return a new state that is created by taken a given action on a given board.

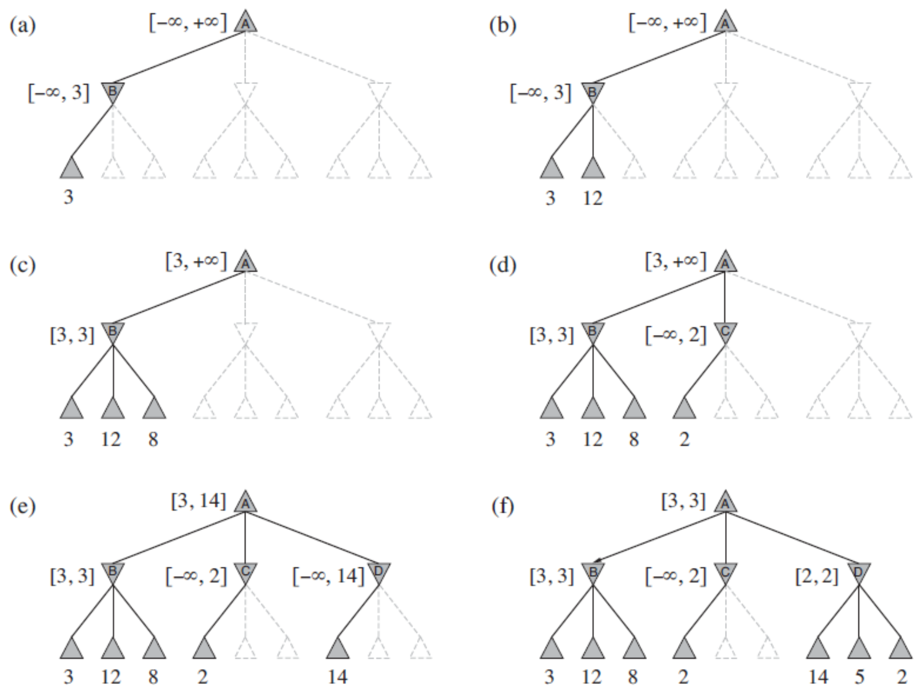
(4) Terminal-Test(s). Similar with that in Part I, this function will tell whether the game is over. However, the standard is slightly different. The one who first win in a single board win the whole game. However, if one of the boards is full but no one wins in that board, the game will go on.

(5) Utility(s). This function will return 1, -1 and 0 respectively in situation that computer wins, player wins and draw.

2. Algorithms

Different from the basic tic-tac-toe, which is not complicated in rules and have a maximal depth of only 9, this 3*3 TTT game is much more difficult to search for an action for the AI agent. Thus, the pure Minimax algorithm will be infeasible to explore all the possible states. Instead, the alpha-beta pruning and H-Minimax algorithm need to be applied to this problem.

The alpha-beta pruning is generally based on the assumption of the Minimax Algorithm that both the players will always choose the optimal move. In this algorithm we introduce two parameters alpha and beta, where alpha is the value of the choice with highest utility score we have found so far at any choice point, while beta is the value of the choice with the lowest utility score. If we dynamically maintain and update these two values every time we explore a state with a smaller Min value or with a bigger Max Value, we can gradually narrow down the range to choose. Therefore, if a state has a successor state with a value out of the range, then the state together with all its successors can be pruned because exploring them makes no sense. In this way, the alpha-beta algorithm reduced a huge amount of work.



The H-Minimax algorithm is typically a kind of depth-limited search algorithm. It is especially useful in this advanced TTT game because the depth of this game is not practical for a pure depth-first search which is used in the basic TTT game. To avoid spending too much time, this algorithm will cut off the search when the depth reaches a certain value. Then, it will use a heuristic eval function to give the current state an estimated value. Here is how it works:

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

3. Code Implementation

The general work flow of this program is similar with that of Part I except the action_search() function. In the new version, the action search applies alpha-beta pruning and H-Minimax algorithm.

To achieve alpha-beta pruning, I just added two more parameters “mini” and “maxi” to Min_value() function and Max_value() function to record the current range of legal values. If a successor of a state has a value out of this range, then other successors of this state won’t be considered, and the program directly enters the next iteration.

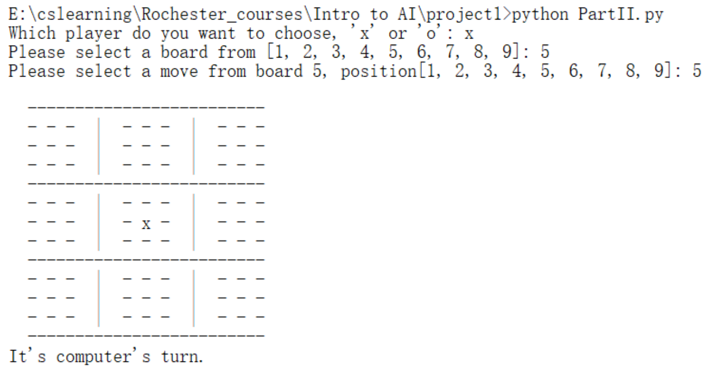
When it comes to the H-Minimax algorithm, I added another parameter “d”. For convenience, I only added it to the Max_value() function which will take the states where it’s the computer’s turn to move. The “d” is the depth of the search and it will increment by one each time Min_value() or Max_value() is called. When the “d” reaches a set value, the search will stop and the current state will be sent to the eval() function. The eval() function will then give an estimated value of this state based on its evaluation system.

The evaluation system of eval() function is designed as follow:

(1) Consider the corner case. If the board corresponding to the last move is full, then computer can play on any board. The program will check all the boards to find whether there is one where the computer is just one step from the win. If there is, it means that computer must win in the next step, so the function will return 1. If there’s no such board, then it means the computer is probably in a bad situation. The program will count the number of boards where the player is one step from the win and give a negative score depending on the number.

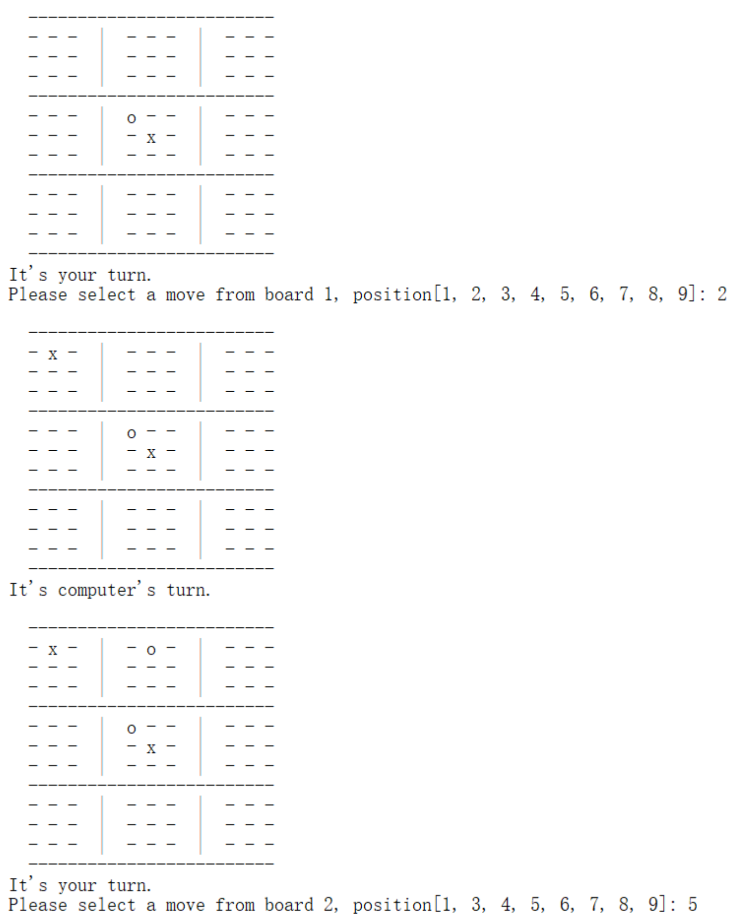
(2) If the board corresponding to the last move is not full, then the computer’s choices are much limited. If in this board it’s one step from win, then the function returns 1. Otherwise, it will go through all the possible moves and divide them into three groups: “dead”, “free”, “other”. A “dead” move is a move that will lead the opponent to the board where it is one step from the win; A “free” move will lead the opponent to a full board so that it can play anywhere in the next move; A “other” move, is a move rather than the former two kinds of moves. After grouping, I gave each group of moves an efficient, and count the weighted average score as the final score of the state.

4. Demo



```
E:\cslearning\Rochester_courses\Intro to AI\project1>python PartII.py
Which player do you want to choose, 'x' or 'o': x
Please select a board from [1, 2, 3, 4, 5, 6, 7, 8, 9]: 5
Please select a move from board 5, position[1, 2, 3, 4, 5, 6, 7, 8, 9]: 5

  -------------------------
  - - -  |  - - -  |  - - -
  - - -  |  - - -  |  - - -
  - - -  |  - - -  |  - - -
  -------------------------
  - - -  |  - - -  |  - - -
  - - -  |  - x -  |  - - -
  - - -  |  - - -  |  - - -
  -------------------------
  - - -  |  - - -  |  - - -
  - - -  |  - - -  |  - - -
  - - -  |  - - -  |  - - -
  -------------------------
It's computer's turn.
```



```
  -------------------------
  - - -  | - - -  | - - -
  - - -  | - - -  | - - -
  - - -  | - - -  | - - -
  -------------------------
  - - -  | o - -  | - - -
  - - -  | - x -  | - - -
  - - -  | - - -  | - - -
  -------------------------
  - - -  | - - -  | - - -
  - - -  | - - -  | - - -
  - - -  | - - -  | - - -
  -------------------------
It's your turn.
Please select a move from board 1, position[1, 2, 3, 4, 5, 6, 7, 8, 9]: 2
  -------------------------
  - x -  | - - -  | - - -
  - - -  | - - -  | - - -
  - - -  | - - -  | - - -
  -------------------------
  - - -  | o - -  | - - -
  - - -  | - x -  | - - -
  - - -  | - - -  | - - -
  -------------------------
  - - -  | - - -  | - - -
  - - -  | - - -  | - - -
  - - -  | - - -  | - - -
  -------------------------
It's computer's turn.
  -------------------------
  - x -  | - o -  | - - -
  - - -  | - - -  | - - -
  - - -  | - - -  | - - -
  -------------------------
  - - -  | o - -  | - - -
  - - -  | - x -  | - - -
  - - -  | - - -  | - - -
  -------------------------
  - - -  | - - -  | - - -
  - - -  | - - -  | - - -
  - - -  | - - -  | - - -
  -------------------------
It's your turn.
Please select a move from board 2, position[1, 3, 4, 5, 6, 7, 8, 9]: 5
```

Part III: Ultimate Tic-Tac-Toe

1. Formulation

The ultimate TTT game is a more advanced version of the advanced TTT. Rather than winning in just one board, the player who want to win the game need to win three boards that art in a horizontal, vertical, or diagonal row. The definition of the problem is slightly changed:

(1) The initial State S_0. In this game the S_0 is similar with that in part II. But I replaced the "nonfull" attribute with "global_state", which records the situation of the global 3*3 grid. Here is what it looks like:

{'player' : [ ], 'comp' : [ ], 'draw' : [ ], 'left' : [1,2,3,4,5,6,7,8,9] }

It's actually a dictionary in python. The keys 'player', 'comp' and 'draw' records sequence numbers of boards that the player wins, the computer wins and draws respectively. The 'left' key store boards where the winner hasn't been decided yet rather than boards that are just full.

(2) Actions are substantially the same with that in Part II.

(3) Result(s,a) will return a new state that is created by taken a given action on a given board.
(4) Terminal-Test(s). The standard of this function has been changed. Only when one of the two players has won three boards that are in a horizontal, vertical, or diagonal row will the game end.
(5) Utility(s). Generally the same with that in Part II, except some small changes in the eval() function.

2. Algorithms

The algorithms used in this game is the same with that in Part II. The eval() function in the H-Minimax algorithm, however, is slightly different.

3. Code Implementation

The work flow of this program is the same with that in Part II.

The check_board() function is the same with the check_state() function in Part II. To do the terminal test, I just sent the global 3*3 grid to be the parameter rather than any of the single board.

I also made some changes in the eval() function. I first considered the corner case that the board corresponding to the last move is full or is won by one of the two players. In this case if there's one board the computer is one step from the win and the board connect other two winning boards in a row, then the function will return 1. If the boards is still available, then the function will check whether in this board the computer is one step from the win and return 1 if the answer is yes. Otherwise, like what it does in the advanced TTT program, it divides the possible moves into four groups. This time the groups are "dead", "free", "finish" and "other". The "dead" moves are the ones that will lead the opponent to the board where it can get the final win in one step; the "free" moves are the ones that will lead the opponent to a unavailable board so that it can make a move on any available boards; the "finish" moves are ones that can help the computer win on that board; the "other" moves are the ones not included in the previous groups. Then according to the numbers of boards that the two player have won and are one step from the win, I gave each group a coefficient and calculated the weighted average score as the return value.

4. Demo

```
It’s your turn.
  ------------------------------------------------------------
Please select a move from board 4, position[1, 2, 8]: 8

  -------------------------
  - - o | x - - | x - -
  - - o | x - - | x o -
  o - o | x - o | - o x
  -------------------------
  - - o | x - o | - - x
  o o x | - x - | x - -
  x o x | - - x | o x -
  -------------------------
  - x - | - - - | x o x
  o o o | x o - | o o o
  x o o | x - - | x - x
  -------------------------
Here is the current global state:
  o x -
  - x -
  o - o

It’s computer’s turn.
  ------------------------------------------------------------

  -------------------------
  - - o | x - - | x - -
  - - o | x - - | x o -
  o - o | x - o | - o x
  -------------------------
  - - o | x - o | - - x
  o o x | - x - | x - -
  x o x | - - x | o x -
  -------------------------
  - x - | x - - | x o x
  o o o | x o - | o o o
  x o o | x - - | x - x
  -------------------------
Here is the current global state:
  o x -
  - x -
  o x o

It’s your turn.
  ------------------------------------------------------------
Computer wins!
```

Evaluation and future work

Here are the testing results of these three games:

1. For the basic TTT, no matter who choose the ‘x’, the computer can ensure a win or a tie game. If the human player makes a mistake, then the computer is bound to win. Otherwise there will be a draw.
2. For the advanced TTT, the results have much to do with the order. If the computer plays first, then it will probably win all the time. If the human player plays first and is smart enough, then it will probably win all the time.

3. For the ultimate TTT, the testing results are similar to the advanced TTT. However, if the human player chooses to play first then it will win in a relatively easier way. That indicate that the eval() function is still far from being perfect.
4. Other problems: The time complexity of Part II and Part III still have a big space to optimize. It may be caused by of the use of python which is slow in recursions or the redundant operations in the search process.

Future work includes designing a more efficient and effective eval() function and reduce the time cost of the programs.

Reference

[1]. Artificial Intelligence: A Modern Method 3rd Edition. Stuart J.Russell, Peter Norvig