

# Tic-Tac-Toe Project Report

Shaojie Wang

September 25, 2017

## 1 Introduction

This project is divide into two parts, normal  $3 \times 3$  tic-tac-toe and advanced tic-tac-toe. Simple minimax algorithm is used on playing  $3 \times 3$  tic-tac-toe, while alpha-beta pruning and heuristic minimax algorithm are used on playing the advanced tic-tac-toe.

In this project, fancy object oriented design is applied to the hole game. Basic member functions are designed based on the content of our textbook (AIMA 5.2-5.4). Combats between agents using different evaluation functions are conducted, and a rather optimized evaluation function is derived eventually.

## 2 Object oriented design

In this section, let's just go through the UML class diagrams of the two classes, shown in table 1 and table 2. The comments after "#" are the usage of these functions or members.

The purpose of such design is to make it convenient to change the rules of the game, rewrite the evaluation function, or construct combats between agents.

## 3 Game demo

In the *\_init\_* function, the program requires an input of 'x' or 'o' ('x' always being the offensive side). Then call the *play()* function to begin the game. Each time we need to input a position number(in normal  $3 \times 3$  tic-tac-toe) or board number together with position number(advanced tic-tac-toe). Then in the while loop, we can play again if one game is over. The examples are in fig. 1 and fig. 2 (fig. 2 drops some moves to save space for this document).

```

[swang115@cycle2 ai]$ python3 ttt.py
Choose your side ('x' or 'o'): x
- - -
- - -
- - -
Input your move: 1
x - -
- - -
- - -
Computer's move: 5
x - -
- o -
- - -
Input your move: 4
x - -
x o -
- - -
Computer's move: 7
x - -
x o -
o - -
Input your move: 6
x - -
x o x
o - -
Computer's move: 3
x - o
x o x
o - -
You lose!
Choose your side ('x' or 'o'):

```

Figure 1: Demo of  $3 \times 3$  tic-tac-toe.

```

[swang115@cycle2 ai]$ python3 attt.py
Choose your side ('x' or 'o'): x
- - -|- - -|- - -
- - -|- - -|- - -
- - -|- - -|- - -
-----
- - -|- - -|- - -
- - -|- - -|- - -
- - -|- - -|- - -
-----
- - -|- - -|- - -
- - -|- - -|- - -
- - -|- - -|- - -
-----
Input your move: 1 1
x - -|- - -|- - -
- - -|- - -|- - -
- - -|- - -|- - -
-----
- - -|- - -|- - -
- - -|- - -|- - -
- - -|- - -|- - -
-----
- - -|- - -|- - -
- - -|- - -|- - -
- - -|- - -|- - -
-----
Computer's move: 1 2
x o -|- - -|- - -
- - -|- - -|- - -
- - -|- - -|- - -
-----
- - -|- - -|- - -
- - -|- - -|- - -
- - -|- - -|- - -
-----
Input your move: 2 1
x o -|x - -|- - -
- - -|- - -|- - -
- - -|- - -|- - -
-----
- - -|- - -|- - -
- - -|- - -|- - -
- - -|- - -|- - -
-----
- - -|- - -|- - -
- - -|- - -|- - -
- - -|- - -|- - -
-----
Input your move: 4 1
x o -|x - -|- - -
o - -|- - -|- - -
- - -|- - -|- - -
-----
x - -|- - -|- - -
- - -|- - -|- - -
- - -|- - -|- - -
-----
- - -|- - -|- - -
- - -|- - -|- - -
- - -|- - -|- - -
-----
Computer's move: 1 6
x o -|x - -|- - -
o - o|- - -|- - -
- - -|- - -|- - -
-----
x - -|- - -|- - -
- - -|- - -|- - -
- - -|- - -|- - -
-----
- - -|- - -|- - -
- - -|- - -|- - -
- - -|- - -|- - -
-----
Input your move: 6 1
x o -|x - -|- - -
o - o|- - -|- - -
- - -|- - -|- - -
-----
x - -|- - -|x - -
- - -|- - -|- - -
- - -|- - -|- - -
-----
- - -|- - -|- - -
- - -|- - -|- - -
- - -|- - -|- - -
-----
Computer's move: 1 5
x o -|x - -|- - -
o o o|- - -|- - -
- - -|- - -|- - -
-----
x - -|- - -|x - -
- - -|- - -|- - -
- - -|- - -|- - -
-----
- - -|- - -|- - -
- - -|- - -|- - -
- - -|- - -|- - -
-----
You lose!
Choose your side ('x' or 'o'): |

```

Figure 2: Demo of advanced tic-tac-toe.

Table 1: UML class diagram of tic\_tac\_toe

<b>tic_tac_toe</b>		
+ s0: list of string	#	Initial state
+ human: string	#	'x' or 'o' for human player
+ pc: string	#	'x' or 'o' for computer player
+ show_once: int	#	Make sure that the same state will be printed only once
- __init__: void	#	Initiate the board
+ player(s): string	#	Return the current player
+ action(s): list of int	#	Return all the possible actions for state s
+ result(s, a): list of string	#	Return the result of action a
+ terminal_test(s): bool	#	Return whether the game is over
+ utility(s, p): int	#	Return the utility value of the state s with player p
+ minimax(s): (int, int)	#	Return the best move for state s and its value
+ show(s): void	#	Print the current board
+ play(): void	#	Start to play the game

Table 2: UML class diagram of advanced\_tic\_tac\_toe

<b>advanced_tic_tac_toe</b>		
+ s0: list of list of string	#	Initial state
+ human: string	#	'x' or 'o' for human player
+ pc: string	#	'x' or 'o' for computer player
+ show_once: int	#	Make sure that the same state will be printed only
+ last_move: (int, int)	#	Record the last move once
- __init__: void	#	Initiate the board
+ player(s): string	#	Return the current player
+ action(s): list of (int, int)	#	Return all the possible actions for state s
+ result(s, a): list of string	#	Return the result of action a
+ terminal_test(s): bool	#	Return whether the game is over
+ evaluation(s): int	#	Return the heuristic utility of the state s
+ alpha_beta_search(s, depth): (int, (int, int))	#	Return the best move for state s and its value
+ max_value(s, alpha, beta, depth): (int, (int, int))	#	Return the best move for state s and its value(for computer)
+ min_value(s, alpha, beta, depth): (int, (int, int))	#	Return the best move for state s and its value(for human)
+ cutoff_test(s): bool	#	Return whether to cut off
+ single_board_over(s): int	#	Return different values, 1 for someone winning, 0 for full, -1 for able to play
+ board_not_full(s): list of int	#	Return the number of boards not full
+ check_siege(s): int	#	Return the number of sieges on a board
+ show(s): void	#	Print the current board
+ play(): void	#	Start to play the game

## 4 Explanation on important code blocks

In this section, I am going to explain the functions of certain code blocks, and how I developed them. Before I go on to the important part, I want to talk about the `stdin`, `stdout` and `stderr` first. At first, the output of `stderr` is very dangling. Sometimes it will appear after `stdout`, sometimes before `stdout`, while the coding sequence is fixed. More strangely, even if I run `stderr.write("something here")` right before `stdin.readline()`, the message "something here" will still show after I finish the `stdin`. Finally I figured this out. This is all about buffering. When `stderr.write("something here")` is executed, `stdin.readline()` will be the next to execute. Then when the `stdin.readline()` starts to expect some input, the output of `stderr.write("something here")` is still not printed because `stdin.readline()` comes so fast. Therefore, only when I finish input, the message of `stderr` will show on screen. To solve this, I simply added `stderr.flush()` right after the `stderr.write("something here")`, so that the message will be forced to show on screen immediately. Same thing is done for `stdout` messages.

### 4.1 Normal $3 \times 3$ tic-tac-toe

For `self.player(self, s)`, I used to think that there should be a `change_player` function. However, inspired by the rule of the game, I just calculated the number of both 'x' and 'o' pieces. Since 'x' always goes first, if 'x' is more than 'o', then it is 'o' moving. Otherwise, if the numbers are equal, then it is 'x' moving.

For the input of human side, I use `try ... except ...` to catch some unexpected troubles instead of writing a lot of `whiles` and `ifs`.

For `self.show(self, s)`, I make it print the board for only once if the player is asked to input again. The tricky part is to set a member variable `show_once`, and every time we call `self.player(self, s)`, we check the value of `show_once`, if it is 0, which means not even shown once, then print and assign it with 1, which means already shown once.

For `self.minimax(self, s)`, the coding is quite easy, because our textbook has already provided us with neat pseudo codes. However, for the utility function, there are more stories to tell about. At first, I used 1 for win, -1 for lose, and 0 for tie. Truly I cannot win a single game since the minimax function works very well. However, the computer may sometimes play some "stupid" moves. In the example displayed in fig. 3 compared to fig. 2, we can see that the computer is only one step to win (after I played 6), but it plays another move (position 2, as the red arrow points) rather than the winning move (3), yet the computer wins eventually. Then I figured out that I did not implement the step cost, which is 1 for each step. After adding this feature into my utility function, the computer plays more "intelligently", which is make a winning move as soon as possible.

```

[swang115@cycle2 ai]$ python3 ttt.py
Choose your side ('x' or 'o'): x
- - -
- - -
- - -
Input your move: 1
x - -
- - -
- - -
Computer's move: 5
x - -
- o -
- - -
Input your move: 4
x - -
x o -
- - -
Computer's move: 7
x - -
x o -
o - -
Input your move: 6
x - -
x o x
o - -
Computer's move: 2
x o -
x o x
o - -
Input your move: 3
x o x
x o x
o - -
Computer's move: 8
x o x
x o x
o o -
You lose!
Choose your side ('x' or 'o'):

```




Figure 3: "Stupid" move of  $3 \times 3$  tic-tac-toe.

An important thing is that in Python, simple assignment is only a process of sticking tags, rather than copy the real content of a variable. When I was testing the program, the board always filled itself up very quickly. Then I noticed that when I tested each move, the board would not return to what it was, because there was only one board with many different tags. The solution is to use deepcopy function or slice (my choice is slice, because it is neat and I do not need to import an extra package).

## 4.2 Advanced tic-tac-toe

The basic codes are mostly the same as those in normal  $3 \times 3$  tic-tac-toe games. The really disturbing part is the rule of the game. Fortunately, I designed the class properly, so I just need to adjust the action function and play function to implement new rules.

For the alpha-beta search function, I made some slight adjustment to the pseudo code on the text book so that I can implement the heuristic function smoothly. I added a parameter, *depth*, to the function, so that I can limit the depth of DFS. I also expand the *max()* and *min()* functions in order to keep track on the move whose heuristic value is optimal (maximum of minimum). The heuristic function, the most important part of advanced tic-tac-toe, will be discussed in the next section.

Note that there are 9 different  $3 \times 3$  tic-tac-toe boards in this game, way of counting the number of 'x' and 'o' can be a little bit different. Here, I use the reduce function, which can be used for adding up all the values in a list. This is very handy, because I can count the number in each of the 9 board, and then call reduce function to obtain the final count.

## 5 Explanation on heuristic function

In this section, I am going to explain the details in evaluation(heuristic) function, and how the results of combats between agents affect the construction of our evaluation function.

When the *cutoff\_test(self, s, depth)* returns true, we need to evaluate the current situation and return a heuristic value that can lead the player to victory. The design of the evaluation rules is tricky. My evaluation function is based on scoring. Here are my ideas.

- (A) Inspired by playing normal  $3 \times 3$  tic-tac-toe that the center point is very important, I marked some special points on the board. They are fixed points and center points. Fixed points are those like (1,1), (4,4), whose board number and position number are the same. Center points are of course the center point of each board.

- (B) Since we want the computer to win, I made the rule such that if the computer plays on the special points, add 1 point; if human plays on the special points, minus 1 point.
- (C) If the computer wins, add 30 points; if human wins, minus 30 points.
- (D) If computer makes a siege<sup>1</sup>, add 3 points.
- (E) If the computer makes a move that leads human to a full board, minus 10 points, because then human can play randomly, which is a big disadvantage for computer player. Vice versa.

Of course, these are only ideas. The efficiency of these rules are then tested through combats between agents with different evaluation functions. It turns out that idea (C, D, E) are all very critical criteria through comparisons. Idea (B) is hard to decide. Here is how I solved this problem. In 3, I conducted 16 different combats among 4 different evaluation functions, and on both offensive and defensive sides. It turned out that evaluation function with fixed points and center points wins most offensive-side games. For defensive-side games, the four evaluation functions seem to perform equivalently. Therefore, for offensive side, I use evaluation function with fixed points and center points, and for defensive side, I can use any of the four functions. Here, for convenience, I just use the same evaluation function as on offensive side. This will reduce the probability that the evaluation function might overestimate the situation. The eventual evaluation function is quite promising. At least I cannot beat the computer with such function.

Table 3: Combat results of different evaluation functions

		Offensive			
		np	c	f	fc
Defensive	np	win	lose	lose	win
	c	lose	lose	win	win
	f	lose	win	lose	win
	fc	win	lose	win	lose

np: no fixed points or center points;

c: with center points only;

f: with fixed points only;

fc: with both fixed and center points.

Note that the depth of search is also a key component. In this case, if we set the depth to 7, it starts to jam when running, so I set the depth to 5, since 5 and 6 perform with no obvious difference.

---

<sup>1</sup>A siege is a situation where the opponent has two pieces in a row, and you place your move on the only rest space to stop the opponent from winning.



## 6 Future works

The future works for this project include fancy GUI for the game, better pruning strategies, and more sophisticated heuristic functions. With better pruning strategy, we can search deeper to increase the probability of winning the game. With more sophisticated heuristic functions, we can make more precise prediction of the probability of winning, and try to achieve the goal of "never overestimate". More combats between agents could be a way. Also, the value of points are not evaluated in my project. For example, when the computer wins, I give it 30 points. Will 50 points be better? This needs further test.