Writeup

Jiayi Qi

netID: jqi8

Part 1. Basic Tic-Tac-Toe

A. Abstraction of Tic-Tac-Toe

Programming Language: Python

State: use a dictionary to represent current state.

{1:1, 2:2, 3:3, 4:4, 5:5, 6:6, 7:7, 8:8, 9:9, "next":"x"}

This is the initial state. The integer keys represent 9 spaces on the board and "next" means which tile goes next. The default values for every space is the location number for displaying the board easier. Whenever a player take a move will change the value of that position into the tile he selected.

Action: use a list to represent actions.

['1','2','3','4','5','6','7','8','9']

Each element represents a move on specific position.

ACTION: use a list to store the applicable actions now. Every time a position is occupied, that position will be removed from actions list.

RESULT: Every move on position *i* will replace the specific value of key *i* with the tile goses next and change the value of key "next" into the mark of the other tile. Meanwhile, remove the move from now_action list.

Cost: In this program, cost doesn't matter.

Initial State: {1:1, 2:2, 3:3, 4:4, 5:5, 6:6, 7:7, 8:8, 9:9, "next":"x"}

Goal State: use a win_state list to store all the possible combination of positions. If one of the positions combination belongs to a same player, then game ends.

[[1,2,3],[4,5,6],[7,8,9],[1,4,7],[2,5,8],[3,6,9],[1,5,9],[3,5,7]]

B. Algorithm

I use the original minimax algorithm. Because the search space is not very big and searching time is acceptable. This algorithm will get the optimal move in every state.

C. Function Introduction

1. *playermove(now_state, now_action)*

   This is the function for human player to enter their move. There is a while loop to re-enter the input if one input is invalid. To check the validity, it will determine whether the input is in *now_action* (a list represent all applicable actions).

2. *printchecker(now_state)*

   This is the function to display the checker of current state.

3. *referee(now_state)*

   This is the function to determine whether the game ends or not. If it ends, return the result (Computer wins, Player wins, draw!). If not, return 0.

4. *computermove(now_state, now_action)*

   This is the function for computer to compute and execute the move. In this function, it calls the function *minimax_decision2* to get the optimal move and time of computing. Also, this function will display the max minimax value, searching time and the move computer took.

5. *max_value(now_state, now_action)*

   This is the function for max level in minimax algorithm. It will return the max value of next level, which is a min level. In this function, it called *min_value* function.

6. *min_value(now_state, now_action)*

   This is the function for min level in minimax algorithm. It will return the min value of next level, which is a max level. In this function, it called *max_value* function.

7. *minimax_decision2(now_state, now_actions)*

   This is the function of getting the action with max minimax value in current state. It will return the optimal action and searching time.

D. Implement

The whole main part for this program is in a while loop to make it restart when last game ends. So the first step in this loop is to initiate the state, actions.
The second step is also a while loop to get the valid selection (upper and lower case of x, o) of tiles.

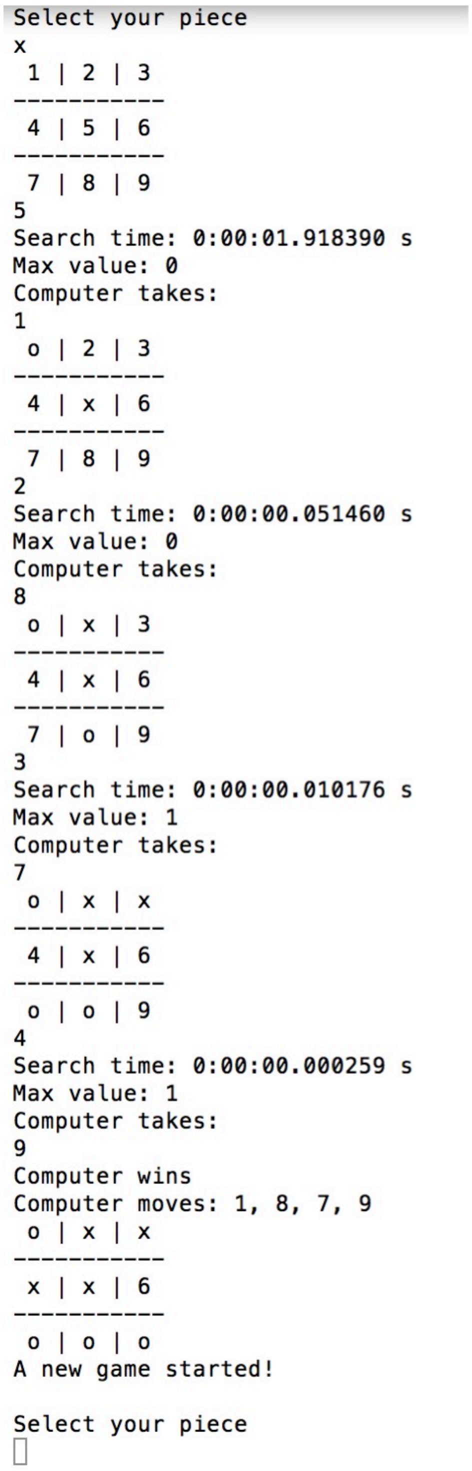
The third step is a while loop too. Every run of this loop is for a player to move. If next player is computer, then call the *computermove* function. If not, call the *playermove* function. After every moves, it will call the *referee* function to determine continue or not. Also, it will call the *printchecker* function to show the current state of the checker for human player to know what move to take.

E. Conclusion

At begin, I think the program will easily beat me in every game. But actually, it can't. The result is that this program will never lose a single game. But if I made a stupid move (not very stupid), it can easily beat me.

As for the searching time, if I select 'x' tile, the first searching time is about 1.8s, and it will decrease with the process of the game because of the reducing search space. If I select ‘o’ tile, the first searching time is about 18s, and it will decrease with the process of the game. The longer first searching time is because of a much bigger searching space. Because there are 9 possible first move, so we can say that the searching space if 9 times the searching space of the first case. That’s reason why the searching time is approximately 9 times the first case searching time.

F. Screenshot



```
Select your piece
x
 1 | 2 | 3
-----------
 4 | 5 | 6
-----------
 7 | 8 | 9
5
Search time: 0:00:01.918390 s
Max value: 0
Computer takes:
1
 o | 2 | 3
-----------
 4 | x | 6
-----------
 7 | 8 | 9
2
Search time: 0:00:00.051460 s
Max value: 0
Computer takes:
8
 o | x | 3
-----------
 4 | x | 6
-----------
 7 | o | 9
3
Search time: 0:00:00.010176 s
Max value: 1
Computer takes:
7
 o | x | x
-----------
 4 | x | 6
-----------
 o | o | 9
4
Search time: 0:00:00.000259 s
Max value: 1
Computer takes:
9
Computer wins
Computer moves: 1, 8, 7, 9
 o | x | x
-----------
 x | x | 6
-----------
 o | o | o
A new game started!

Select your piece
```

Part 2. 9 Boards Tic-Tac-Toe

A. Abstraction of 9 Board Tic-Tac-Toe

Programming language: Python

States: use a list contained 10 elements to represent the states. All the elements are dictionaries. Each of first 9 dictionaries is a basic TTT board like the one in part 1. The index of the dictionary plus 1 represents the position of the board. Last dictionary has three keys, *next*, *nextpos* and *full*. The value of *next* means the next player to move, the value of *nextpos* means next move should be on which board/boards. And *full* means which board/boards is/are full.

Action: use a list contained 9 elements, each of them is a list represents the actions on the specific board. So the index plus 1 is the board number.

ACTION: use a same list as Action to store all the applicable actions now. Every time taking a move will remove the move from this list.

RESULT: Every move *ij* will replace the specific value of key *j in* the *i*th dictionary with the tile goes next and change the value of key "next" into the mark of the other tile. Meanwhile, remove the move from naction list. Also, the *nextpos* value will be change to *j*.

Initial State: Every board is clear, *next* value is *x*, *nextpos* value is *[1,2,3,4,5,6,7,8,9]* and *full* value is *[]*.

Goal State: Any single board ends means the whole game ends.

B. Algorithm

I use Alpha-Beta-Pruning minimax algorithm along with depth limit search with a heuristic function. Since the state space and search space is very large so that the original minimax algorithm is not applicable anymore. Because the shallowest win for this search tree is 5 so I make the depth limit 5.

As for the heuristic function, let's say x is computer. Then in a single board, there is two x in a single row/column/diagonal and no o in this row/column/diagonal, the heuristic value will plus 2. If two 0 in a single row/column/diagonal and no x in this row/column/diagonal, the heuristic value will minus 2. If there are three x in a single row/column/diagonal, the value will plus 100. And if there are three o in a single row/column/diagonal, the value will minus 100. The reason why this value is much bigger than the 2 adjacent tile is that I want to make the award for win much bigger so that the computer will tend to take the winning move.

C. Function Introduction

1. *playermove(nstate, naction)*

   This is the function for human player to enter their move. There is a while loop to re-enter the input if one input is invalid. To check the validity, move *ij* will make me to check whether *j* is in the *i*th list in *naction*. First argument is current state. Second one is current applicable actions. If *j* in the list of 'full', the *nextpos* value will be set to all the unfinished board. Otherwise the *nextpos* value will be *j.*

2. *computermove(nstate, naction)*
   This is the function for computer to compute and execute the move. In this function, it calls the function *absearch* to get the move.

3. *displaychecker(nstate)*
   This is the function to display the checker of current state.

4. *referee(now_astate)*
   Functionality is the same as Part 1. But in this case, it will only determine the game status in a single board.

5. *headreferee(nstate)*
   Determine the whole game status bases on *referee* function. It will call *referee* function for every board. If all the 9 board draw, it will return a draw result. If a single board ends with a winner, it will return the result. Otherwise it will continue the game. Also this function will add the full board number into the value of 'full'.

6. *max_value(nstate,a,b,naction,limit)*
   The function for max level Alpha-Beta-Search. In this function, it will call *min_value* function. If it gets to a cutoff condition (limit depth or game ending), it will return the heuristic value of that state.

7. *min_value(nstate,a,b,naction,limit)*
   The function for min level Alpha-Beta-Search. In this function, it will call *max_value* function. If it gets to a cutoff condition (limit depth or game ending), it will return the heuristic value of that state.

8. *absearch(nstate, naction)*
   The function for getting the action with the max heuristic value.

9. *heuristic(nstate)*
   The function for computing the heuristic value of *nstate* (a state). The computing strategy has been introduced in Part 2.B.

10. *getaction(nstate, naction)*
    This function is for get all applicable actions from *naction*. The reason why I wrote this function is that some time the *nextpos* have multiple values so that it is hard to use the value as the index to get the applicable action list.

D. Implement

The whole main part for this program is in a while loop to make it restart when last game ends. So the first step in this loop is to initiate the state, actions.

The second step is also a while loop to get the valid selection (upper and lower case of x, o) of tiles.

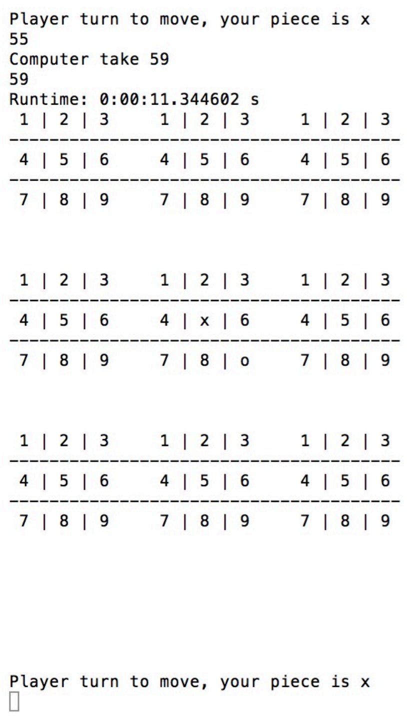
The third step is a while loop too. Every run of this loop is for a player to move. If next player is computer, then call the *computermove* function and display the searching time. If not, call the *playermove* function. After every moves, it will call the head*referee* function to determine continue or not. Also, it will call the *displaychecker* function to show the current state of the checker for human player to know what move to take.

E. Conclusion

This heuristic is admissible. The program shows an ability to win this game.

When the limited depth is 5, if the computer use o tile, the average searching time is about 6s. If the computer takes the first move, the first search time will be around 60s and the subsequent searching time will have an average of 6s. The reason is that the first searching space is much bigger than the first case. However, every time a single board meets a draw, there will be an about 50s searching time for once. This is because of after an end on a single board, there *nextpos* value will be set to all the unfinished board. This will increase the state space.

F. Screenshot



```
Player turn to move, your piece is x
55
Computer take 59
59
Runtime: 0:00:11.344602 s
 1 | 2 | 3     1 | 2 | 3     1 | 2 | 3
---------------------------------------
 4 | 5 | 6     4 | 5 | 6     4 | 5 | 6
---------------------------------------
 7 | 8 | 9     7 | 8 | 9     7 | 8 | 9


 1 | 2 | 3     1 | 2 | 3     1 | 2 | 3
---------------------------------------
 4 | 5 | 6     4 | x | 6     4 | 5 | 6
---------------------------------------
 7 | 8 | 9     7 | 8 | o     7 | 8 | 9


 1 | 2 | 3     1 | 2 | 3     1 | 2 | 3
---------------------------------------
 4 | 5 | 6     4 | 5 | 6     4 | 5 | 6
---------------------------------------
 7 | 8 | 9     7 | 8 | 9     7 | 8 | 9




Player turn to move, your piece is x
```

Part 3. Ultimate Tic-Tac-Toe

A. Abstraction of Ultimate Tic-Tac-Toe

States, **Action**, **ACTION**, **RESULT** and **Initial state** are the same as 9 Board Tic-Tac-Toe. The only different thing is the **goal state**.

Goal State: same player wins 3 child boards in row horizontally, vertically or diagonally.

B. Algorithm

Use Alpha-Beta-Pruning minimax algorithm along with depth limit and heuristic. Because the state space is to large to implement an original minimax algorithm.

As for heuristic, based on the 9 board Tic-Tac-Toe heuristic, I added more award after winning a single board. If a move on adjacent board of the board already win, the growth rate (used to be 2 or -2) will be doubled (now is 4 or -4). If the board is not adjacent with the win board, the growth rate will remain 2 or -2. Also the award of winning of adjacent board is doubled. Not adjacent board remains 100 and -100. In this case the computer will tend to win and move on the adjacent board.

C. Function Introduction

Most of the functions are the same as 9 Board TTT. The only two we need to modify is *headreferee* function and *heuristic* function. The previous one works as a referee to determine whether the game ends. If the rule changes, the referee should be change. The latter one works like a evaluation for the computer so that the rule changes would result in the evaluation change.

1. *headreferee(nstate)*

   The differences between this version and the one in 9 Board TTT is that the game ends when same player wins 3 child boards in row horizontally, vertically or diagonally. So I change the end condition. So the board number which one player wins or meets a draw will be added to 'full' list. This function will call *referee* in every board, so that I store the number of board each player wins in two different list. And then I check the list with a length greater than 2 (have a chance to win) whether there is a horizontal, vertical or diagonal. If it does, then return the result.

2. *heuristic*(nstate)

   The updated strategy has been mentioned before. To implement it, before we plus a value, I added a condition statement to check if this board is near a win board. I use some matrix transform to get a list which contains all the adjacent position of the board player or computer already wins.

D. Implement

The whole main part for this program is in a while loop to make it restart when last game ends. So the first step in this loop is to initiate the state, actions.

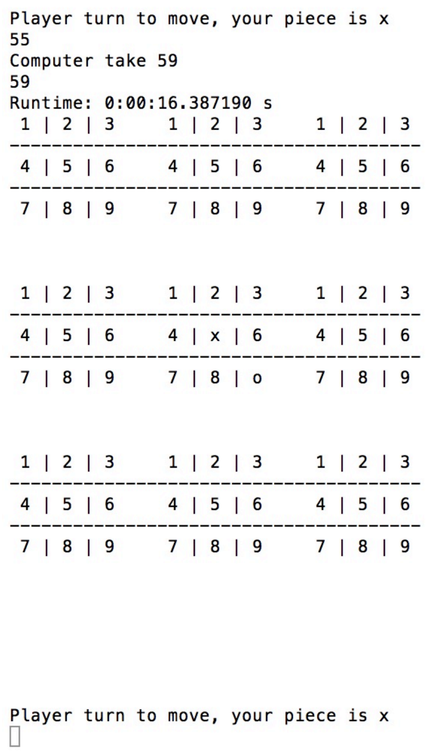
The second step is also a while loop to get the valid selection (upper and lower case of x, o) of tiles.

The third step is a while loop too. Every run of this loop is for a player to move. If next player is computer, then call the *computermove* function and display the searching time. If not, call the *playermove* function. After every moves, it will call the *headreferee* function to determine continue or not. Also, it will call the *displaychecker* function to show the current state of the checker for human player to know what move to take.

E. Conclusion

This program has the ability to win an ultimate 9 board TTT. And the searching time is approximately the same as 9 board TTT because the algorithm and searching space is almost the same. The only two different things are the ending condition and heuristic. Both of them have little time complexity influences on the whole program. However, every time a single board meets a draw or one of the player win, there will be an about 50s searching time for once. This is because of after an end on a single board, there *nextpos* value will be set to all the unfinished board. This will increase the state space.

F. Screenshot



```
Player turn to move, your piece is x
55
Computer take 59
59
Runtime: 0:00:16.387190 s
 1 | 2 | 3     1 | 2 | 3     1 | 2 | 3
----------------------------------------
 4 | 5 | 6     4 | 5 | 6     4 | 5 | 6
----------------------------------------
 7 | 8 | 9     7 | 8 | 9     7 | 8 | 9


 1 | 2 | 3     1 | 2 | 3     1 | 2 | 3
----------------------------------------
 4 | 5 | 6     4 | x | 6     4 | 5 | 6
----------------------------------------
 7 | 8 | 9     7 | 8 | o     7 | 8 | 9


 1 | 2 | 3     1 | 2 | 3     1 | 2 | 3
----------------------------------------
 4 | 5 | 6     4 | 5 | 6     4 | 5 | 6
----------------------------------------
 7 | 8 | 9     7 | 8 | 9     7 | 8 | 9




Player turn to move, your piece is x
```

I would like my program to enter the tournament.