Tic-tac-toe Documentation

Yaxin Huang
elfin@ccs.neu.edu

The program implements Minimax algorithm with alpha-beta pruning, as the Figure 5.7 in our textbook shows:

```
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 5.7 The alpha–beta search algorithm. Notice that these routines are the same as the MINIMAX functions in Figure 5.3, except for the two lines in each of MIN-VALUE and MAX-VALUE that maintain α and β (and the bookkeeping to pass these parameters along).

The reason I chose this algorithm is that this version of Tic-tac-toe will only need a 3*3 grid to play the game. That is to say, the space and time we need to search the whole state space will not be too large. Also, with pruning, it is easier to implement, in comparison with heuristic methods, and it is efficient enough to figure out the next move within time limit. Running on my computer(Intel T6600 2GHz, 2GB memory) only cost no more than 1 second to respond.

There is a class called **Grid**, each instance of it will have these variables:
data -- an array(list) to represent the grid.
aiSign -- the notation used by the AI.
playerSign -- the notation used by the player.
And in the class Grid, it mainly provides these methods:
legal_move_judge(self,move)
-- Given the data, judge if the given move is legal.
legal_move_list(self)

-- Return the list of legal moves based on the data.

terminal_test(self)

-- Based on the data, judge if this is a terminal state.

There is another class called State. When each player makes a move, a new state will be generated to represent the game state. And the AI also use instances of this class to do alpha-beta-search.

Each instance of class State will have the following variables:

moveval -- a dictionary, initially containing all the legal moves. After AI has done the alpha-beta-search, the corresponding utility value will be filled.

grid -- an instance of class Grid. Represent the grid in this state.

lastTurn -- whose turn it was in the last turn.

And in the class State, it mainly provides these methods:

get_legal_moves(self)

-- return all the legal moves of this state.

legal_move_judge(self,move)

-- judge if the given move is legal.

result(self,move)

-- return the next state of this state after the given move.

ab_search(self)

-- return the move with the best value v.

It will call the function max_value(state,a,b) and then max_value will call the function min_value(state,a,b). Just as the algorithm in Figure 5.7. One different is that, every v calculated for a move will be stored in the dictionary “moveval”.

terminal_test(self)

-- judge if the game reaches the terminal state.

-- return 0: The game continues.

-- return 1: AI wins.

-- return 2: The player wins.

-- return 3: Draw.

There is also a function utility(state) define in the file absearch.py. It is called when a terminal state is reached and the utility value of this state will be returned.

When a game is started, the program will randomly choose AI or the player to go first, and then an initial state will be created. After each player(AI/human) has made a move, the state will evolve to the next one and it’s now the other player’s turn to move. The result will be printed out after it reaches a terminal state.