Lab 4 Gaming and Minimax Algorithms

SUSTC

Minimax Decision

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

Minimax Decision

```
def minimax_decision(state, game):
    """Given a state in a game, calculate the best move by searching
    forward all the way to the terminal states. [Figure 5.3]"""

    player = game.to_move(state)

    def max_value(state):
        if game.terminal_test(state):
            return game.utility(state, player)
        v = -infinity
        for a in game.actions(state):
            v = max(v, min_value(game.result(state, a)))
        return v

    def min_value(state):
        if game.terminal_test(state):
            return game.utility(state, player)
        v = infinity
        for a in game.actions(state):
            v = min(v, max_value(game.result(state, a)))
        return v

    # Body of minimax_decision:
    return argmax(game.actions(state),
                  key=lambda a: min_value(game.result(state, a)))
```

Tic Tac Toe Game 1/2

```
class TicTacToe(Game):
    """Play TicTacToe on an h x v board, with Max (first player) playing 'X'.
    A state has the player to move, a cached utility, a list of moves in
    the form of a list of (x, y) positions, and a board, in the form of
    a dict of {(x, y): Player} entries, where Player is 'X' or 'O'."""

    def __init__(self, h=3, v=3, k=3):
        self.h = h
        self.v = v
        self.k = k
        moves = [(x, y) for x in range(1, h + 1)
                 for y in range(1, v + 1)]
        self.initial = GameState(to_move='X', utility=0, board={}, moves=moves)

    def actions(self, state):
        "Legal moves are any square not yet taken."
        return state.moves

    def result(self, state, move):
        if move not in state.moves:
            return state  # Illegal move has no effect
        board = state.board.copy()
        board[move] = state.to_move
        moves = list(state.moves)
        moves.remove(move)
        return GameState(to_move=('O' if state.to_move == 'X' else 'X'),
                         utility=self.compute_utility(board, move, state.to_move),
                         board=board, moves=moves)

    def utility(self, state, player):
        "Return the value to player; 1 for win, -1 for loss, 0 otherwise."
        return state.utility if player == 'X' else -state.utility
```

Tic Tac Toe Game 2/2

```
def terminal_test(self, state):
    "A state is terminal if it is won or there are no empty squares."
    return state.utility != 0 or len(state.moves) == 0

def display(self, state):
    board = state.board
    for x in range(1, self.h + 1):
        for y in range(1, self.v + 1):
            print(board.get((x, y), '.'), end=' ')
        print()

def compute_utility(self, board, move, player):
    "If 'X' wins with this move, return 1; if 'O' wins return -1; else return 0."
    if (self.k_in_row(board, move, player, (0, 1)) or
            self.k_in_row(board, move, player, (1, 0)) or
            self.k_in_row(board, move, player, (1, -1)) or
            self.k_in_row(board, move, player, (1, 1))):
        return +1 if player == 'X' else -1
    else:
        return 0

def k_in_row(self, board, move, player, delta_x_y):
    "Return true if there is a line through move on board for player."
    (delta_x, delta_y) = delta_x_y
    x, y = move
    n = 0  # n is number of moves in row
    while board.get((x, y)) == player:
        n += 1
        x, y = x + delta_x, y + delta_y
    x, y = move
    while board.get((x, y)) == player:
        n += 1
        x, y = x - delta_x, y - delta_y
    n -= 1  # Because we counted move itself twice
    return n >= self.k
```

α-β full search

```python
def alphabeta_full_search(state, game):
    """Search game to determine best action; use alpha-beta pruning.
    As in [Figure 5.7], this version searches all the way to the leaves."""

    player = game.to_move(state)

    # Functions used by alphabeta
    def max_value(state, alpha, beta):
        if game.terminal_test(state):
            return game.utility(state, player)
        v = -infinity
        for a in game.actions(state):
            v = max(v, min_value(game.result(state, a), alpha, beta))
            if v >= beta:
                return v
            alpha = max(alpha, v)
        return v

    def min_value(state, alpha, beta):
        if game.terminal_test(state):
            return game.utility(state, player)
        v = infinity
        for a in game.actions(state):
            v = min(v, max_value(game.result(state, a), alpha, beta))
            if v <= alpha:
                return v
            beta = min(beta, v)
        return v

    # Body of alphabeta_search:
    best_score = -infinity
    beta = infinity
    best_action = None
    for a in game.actions(state):
        v = min_value(game.result(state, a), best_score, beta)
        if v > best_score:
            best_score = v
            best_action = a
    return best_action
```

α-β cutting-off search

```
def alphabeta_search(state, game, d=4, cutoff_test=None, eval_fn=None):
    """Search game to determine best action; use alpha-beta pruning.
    This version cuts off search and uses an evaluation function."""

    player = game.to_move(state)

    # Functions used by alphabeta
    def max_value(state, alpha, beta, depth):
        if cutoff_test(state, depth):
            return eval_fn(state)
        v = -infinity
        for a in game.actions(state):
            v = max(v, min_value(game.result(state, a),
                                 alpha, beta, depth + 1))
            if v >= beta:
                return v
            alpha = max(alpha, v)
        return v

    def min_value(state, alpha, beta, depth):
        if cutoff_test(state, depth):
            return eval_fn(state)
        v = infinity
        for a in game.actions(state):
            v = min(v, max_value(game.result(state, a),
                                 alpha, beta, depth + 1))
            if v <= alpha:
                return v
            beta = min(beta, v)
        return v

    # Body of alphabeta_search starts here:
    # The default test cuts off at depth d or at a terminal state
    cutoff_test = (cutoff_test or
                   (lambda state, depth: depth > d or
                    game.terminal_test(state)))
    eval_fn = eval_fn or (lambda state: game.utility(state, player))
    best_score = -infinity
    beta = infinity
    best_action = None
    for a in game.actions(state):
        v = min_value(game.result(state, a), best_score, beta, 1)
        if v > best_score:
            best_score = v
            best_action = a
    return best_action
```

Coding Example

```
from games import (GameState, Game, TicTacToe, random_player, alphabeta_player,
                   alphabeta_search)
import time
```

- Computer players

```
def random_player(game, state):
    "A player that chooses a legal move at random."
    return random.choice(game.actions(state))


def alphabeta_player(game, state):
    return alphabeta_full_search(state, game)


def alphabeta_prune_player(game, state):
    return alphabeta_search(state, game,6)
```

- Human player

```
def human_player(game, state):
    "A human player."
    inputed_num = input("input position in the form of x,y \n")
    my_num=(int(inputed_num[0]),int(inputed_num[2]))
    return my_num
```

Asking for input of the move

Coding Example 1

- Play game

```
def play_game(game, *players):
    """Play an n-person, move-alternating game."""

    state = game.initial
    while True:
        for player in players:
            move = player(game, state)
            state = game.result(state, move)
            #------------------------------
            game.display(state)
            print('\n')
            #------------------------------
            if game.terminal_test(state):
                game.display(state)
                return game.utility(state, game.to_move(game.initial))
```

Display each move

Coding Example 1

- Implementation of human play against computer

```
ttt = TicTacToe()
# 1st ply X

ttt.display(ttt.initial)
print('\n')
play_ttt=play_game(ttt, human_player, alphabeta_player)
print("The result: %s" %(play_ttt))
```

Coding Example 2

- Playing game without printing the board

```
def play_game(game, *players):
    """Play an n-person, move-alternating game."""

    state = game.initial
    while True:
        for player in players:
            move = player(game, state)
            state = game.result(state, move)

            if game.terminal_test(state):
                return game.utility(state, game.to_move(game.initial))
```

Coding Example 2

- Implementation and calculate the performance of alphabeta player

Use the function to record the running time

```
result_all=0
pstart = time.clock()

for i in range(100):
    ttt = TicTacToe()
    play_ttt=play_game(ttt, random_player, alphabeta_prune_player)
    #play_ttt=play_game(ttt, random_player, alphabeta_player)
    print("The result: %s, %s" %(i, play_ttt))
    result_all=result_all+play_ttt

pend = time.clock()
ptime=pend-pstart

print("The winning rate: %s" %(-result_all/100))
print("The running time: %s" %(ptime))
```

Record the score