Artificial Intelligence - Exercise Sheet 5

by Lucas-Raphael Müller

I had the impression that this weeks exercise was quite much and would have been better split over two weeks.

1 Tic Tac Toe

a) General Design

Suitable utility values should agree with our evaluation function. Therefore $u \in \{-1, 0, 1\}$ for a lost, draw or won scenario are reasonable choices. The tree can not get deeper than the number of fields in the tic tac toe case. The depth is therefore 9 (or including the empty board 10). If one does not stop going deeper even when the game is already won, the number of nodes can be calculated by

$$n < \sum_{i=1}^{9} \frac{9!}{i!}. \tag{1}$$

It is not easy (if not not possible) to give a closed form equation excluding the nodes when the game is already won, therefore it's an upper limit.

b) Implementation and Testing

I designed a few test cases to ensure correctly working minimax algorithm and board behaviour. All tesets **passed**. They can be found in 4 a) and can be executed via *python -m pytest test_ board.py* locally or automatically as below. Testing has been done on *Travis CI*.[1]

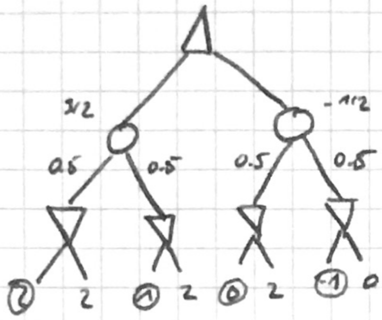
2 Pruning

Correction: It's sufficient to know 1-3 and 5 since the expected value of the left hand side can not be achieved with any number of 6-8, therefore 4 is also not needed.

[1] `https://travis-ci.org/Haydnspass/artificial_intelligence`

Question 2 Pruning

$x_i \in \mathbb{R}$



(a)

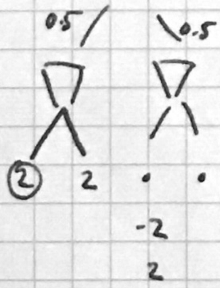
Δ: maximize ∇: minimize

(b)

The values of the leaves 1-6 are not sufficient, since 7th and/or 8th could be both greater than 3. (e.g. +∞, +∞)

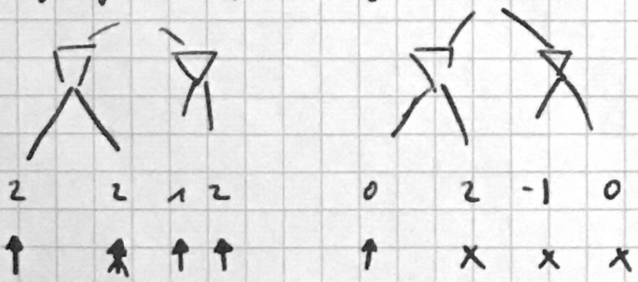
In case we know the 7th leave to be -1, the 8th is not needed anymore since the rational opponent won't choose it.

(c) now restrict ourselfs to $x_i \in [-2, 2]$



the expected value can be between 0 and 2.

(d) by going from left to right:



max

Marks 1-5 since 6-8 have no influence in this domain regime.

3 Four Dices Straight

General Approach

In principle we can form 4 categories of moves, i.e. the different number of dices we roll again. There are

- 4 possibilities to roll 1 dice,
- 6 to roll 2,
- 4 to roll 3,
- 1 to roll all 4.

The total number of outcomes are $6^{n_{redrawn}}$ where many are equal, because one does not care about the order. This could be similiarily implemented as tic tac toe but accounting for a chance game.

4 Appendix: Python Source Code

a) Testing minimax

```
import numpy as np
from tictactoe import Board
from tictactoe import minimax

test_board = list()
result = list()

test_board.append(np.array([['X', 'X', 'X'], ['O',
    'X', 'O'], ['X', 'O', 'X']]))
result.append([])

test_board.append(np.array([['X', 'O', 'X'], ['O', '',
    'O'], ['X', 'O', 'X']]))
result.append([[1, 1]])

def test_full_board():
    t = test_board[0]
    r = result[0]
    b = Board(t)
    assert b.possible_moves() == r


def test_semi_full_board():
    t = test_board[1]
    r = result[1]
    bl = Board(t)
    assert np.array_equal(bl.possible_moves(), r)


test_alg = list()
result_alg = list()

"""
NOTE: The following board configurations must be valid
    in a sense that they are reachable,
since current and next players are determined by the
    board configuration.
"""

test_alg.append(np.array([
```

```
    ['X', '', 'X'],
    ['X', 'O', 'O'],
    ['O', 'O', 'X']]))

result_alg.append([0, 1])


test_alg.append(np.array([
    ['O', '', 'X'],
    ['O', '', 'X'],
    ['', '', '']]))

result_alg.append([2, 0])


test_alg.append(np.array([
    ['O', 'O', 'X'],
    ['O', 'X', 'X'],
    ['', '', '']]))

result_alg.append([2, 0])


def test_minimax():
    for t, r in zip(test_alg, result_alg):
        b = Board(t)
        assert np.array_equal(minimax(b), r)
```

b) Code

```
import numpy as np
from copy import deepcopy

board_rows = 3
board_cols = 3


class Board(object):

    PLAYER_1 = 'O'
    PLAYER_2 = 'X'

    win_count = 3
```

```python
def __init__(self, board=np.empty((board_rows,
        board_cols), dtype=str), computer_player=[]):
    self.board = board
    if computer_player == []:
        self.computer_player = self.next_player
        print('Computer is player: ',
            self.computer_player)
    else:
        self.computer_player = computer_player

def possible_moves(self):
    if self.check_winner(self.PLAYER_1) or
        self.check_winner(self.PLAYER_2):
        return []

    is_empty = np.stack(np.where(self.board ==
        ''), axis=1)
    if is_empty.size == 0:
        return []
    return is_empty

def parse(self, move, player, deep=True):
    if deep:
        cloned_board = deepcopy(self)
        cloned_board.board[tuple(move)] = player
        return cloned_board
    else:
        self.board[tuple(move)] = player

@property
def num_coins(self):
    return np.sum(self.board == self.PLAYER_1) +
        np.sum(self.board == self.PLAYER_2)

@property
def current_player(self):
    if self.next_player == self.PLAYER_1:
        return self.PLAYER_2 # to make player 1
            first player
    else:
        return self.PLAYER_1

@property
def next_player(self):
```

```
    if np.sum(self.board == self.PLAYER_1) >
        np.sum(self.board == self.PLAYER_2):
        return self.PLAYER_2 # to make player 1
            first player
    else:
        return self.PLAYER_1

def utility(self, player, mode='soft'):
    u = 0
    if mode == 'soft':
        for i in range(board_rows):
            u = u + np.sum(self.board[i,:] ==
                player)
        for i in range(board_cols):
            u = u + np.sum(self.board[:, i] ==
                player)
        # maybe add nebendiagonalen later
        u = u + np.sum(np.diag(self.board) ==
            player)
        u = u +
            np.sum(np.diag(np.fliplr(self.board))
            == player)

        if self.check_winner(player):
            u = np.inf
    elif mode == 'hard':
        if self.check_winner(player):
            u = 1

    return u

def check_winner(self, player=[PLAYER_1,
    PLAYER_2]):
    if self.num_coins <= 4:
        return False

    for p in player:
        for i in range(board_rows):
            if np.sum(self.board[i, :] == p) >=
                self.win_count:
                return True
        for i in range(board_cols):
            if np.sum(self.board[:, i] == p) >=
                self.win_count:
```

```python
                    return True
                # maybe add nebendiagonalen later
                if np.sum(np.diag(self.board) == p) >=
                    self.win_count:
                     return True
                if np.sum(np.diag(np.fliplr(self.board))
                    == p) >= self.win_count:
                     return True

        return False


def minimax(graph):

    def min_play(graph):
        if graph.check_winner():
            return graph.utility(graph.computer_player)

        best_score = float('inf')

        for move in graph.possible_moves():
            clone = graph.parse(move,
                graph.next_player,
                graph.computer_player)
            score = max_play(clone)

            if score < best_score:
                best_move = move
                best_score = score

        return best_score

    def max_play(graph):
        if graph.check_winner():
            return graph.utility(graph.computer_player)

        best_score = float(' inf')

        for move in graph.possible_moves():
            clone = graph.parse(move,
                graph.next_player,
                graph.computer_player)
            score = min_play(clone)
```

```python
            if score > best_score:
                best_move = move
                best_score = score

        return best_score

    moves = graph.possible_moves()
    best_move = moves[0]
    best_score = float('inf')
    for move in moves:
        clone = graph.parse(move, graph.next_player,
            graph.computer_player)
        score = min_play(clone)
        if score > best_score:
            best_move = move
            best_score = score
    return best_move


if __name__ == '__main__':
    b = Board(np.array([
    ['X', '', ''],
    ['', 'X', 'O'],
    ['X', 'O', '']]))
    v = minimax(b)
    print(v)
```