Artificial Intelligence 2020/21

Homework 1

Agent and Infrastructure implementation

Stefano Florio 278266

In [66]:
```
from collections import namedtuple, Counter, defaultdict
import random
import math
import functools
```

Source: **aima-python** (https://github.com/aimacode/aima-python) - Python implementation of algorithms from Russell And Norvig's "Artificial Intelligence - A Modern Approach"

Game classes and functions

We define the abstract class `Game` that will hold the data of turn-taking n-player games. Then we define `play_game`, which takes a game of a dictionary of `{player_name: strategy_function}` pairs, and plays out the game checking whose turn it is.

In [67]:
```
class Game:
    """A game has a terminal test and a utility for each terminal state.
    To create a game, subclass this class and implement `actions`, `result`,
    `is_terminal`, and `utility`. You will also need to set the .initial
    attribute to the initial state; this can be done in the constructor."""

    def actions(self, state):
        """Return a collection of the allowable moves from this state."""
        raise NotImplementedError

    def result(self, state, move):
        """Return the state that results from making a move from a state."""
        raise NotImplementedError

    def is_terminal(self, state):
        """Return True if this is a final state for the game."""
        return not self.actions(state)

    def utility(self, state, player):
        """Return the value of this final state to player."""
        raise NotImplementedError
```

In [68]:
```
def play_game(game, strategies: dict, verbose=False):
    """Play a turn-taking game. `strategies` is a {player_name: function} dict,
    where function(state, game) is used to get the player's move."""
    state = game.initial
    while not game.is_terminal(state):
        player = state.to_move
        move = strategies[player](game, state)
        state = game.result(state, move)
        if verbose:
            print('Player', player, 'move:', move)
```

```
            print(state)
    return state
```

Search algorithms

We define some search algorithms. Each takes as input the game we are playing and the current state of the game, returning a `(value, move)` pair, where `value` is the utility that algorithm computes for the player whose turn it is to move, and `move` is the move itself.

First we define `minimax_search`, which exhaustively searches the game tree to find an optimal move (assuming both players play optimally), and `alphabeta_search`, which does the same computation, but prunes parts of the tree that could not possibly have an affect on the optimnal move.

In [69]:

```
infinity = math.inf

def minimax_search(game, state):
    """Search game tree to determine best move; return (value, move) pair."""

    player = state.to_move

    def max_value(state):
        if game.is_terminal(state):
            return game.utility(state, player), None
        v, move = -infinity, None
        for a in game.actions(state):
            v2, _ = min_value(game.result(state, a))
            if v2 > v:
                v, move = v2, a
        return v, move

    def min_value(state):
        if game.is_terminal(state):
            return game.utility(state, player), None
        v, move = +infinity, None
        for a in game.actions(state):
            v2, _ = max_value(game.result(state, a))
            if v2 < v:
                v, move = v2, a
        return v, move

    return max_value(state)
```

In [70]:

```
def alphabeta_search(game, state):
    """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 = state.to_move

    def max_value(state, alpha, beta):
        if game.is_terminal(state):
            return game.utility(state, player), None
        v, move = -infinity, None
        for a in game.actions(state):
            v2, _ = min_value(game.result(state, a), alpha, beta)
            if v2 > v:
                v, move = v2, a
                alpha = max(alpha, v)
            if v >= beta:
                return v, move
        return v, move
```

```python
    def min_value(state, alpha, beta):
        if game.is_terminal(state):
            return game.utility(state, player), None
        v, move = +infinity, None
        for a in game.actions(state):
            v2, _ = max_value(game.result(state, a), alpha, beta)
            if v2 < v:
                v, move = v2, a
                beta = min(beta, v)
            if v <= alpha:
                return v, move
        return v, move

    return max_value(state, -infinity, +infinity)
```

We define `h_alphabeta_search` which combine the usage of a cache function to avoid the computation of already visited states, and the usage of a depth limit. This allow us to execute in reasonable time some game moves computation

In [71]:

```python
def cache1(function):
    "Like lru_cache(None), but only considers the first argument of function."
    cache = {}
    def wrapped(x, *args):
        if x not in cache:
            cache[x] = function(x, *args)
        return cache[x]
    return wrapped
```

In [72]:

```python
def cutoff_depth(d):
    """A cutoff function that searches to depth d."""
    return lambda game, state, depth: depth > d

def h_alphabeta_search(game, state, cutoff=cutoff_depth(6), h=lambda s, p: 0):
    """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 = state.to_move

    @cache1
    def max_value(state, alpha, beta, depth):
        if game.is_terminal(state):
            return game.utility(state, player), None
        if cutoff(game, state, depth):
            return h(state, player), None
        v, move = -infinity, None
        for a in game.actions(state):
            v2, _ = min_value(game.result(state, a), alpha, beta, depth+1)
            if v2 > v:
                v, move = v2, a
                alpha = max(alpha, v)
            if v >= beta:
                return v, move
        return v, move

    @cache1
    def min_value(state, alpha, beta, depth):
        if game.is_terminal(state):
            return game.utility(state, player), None
        if cutoff(game, state, depth):
            return h(state, player), None
        v, move = +infinity, None
```

```python
        for a in game.actions(state):
            v2, _ = max_value(game.result(state, a), alpha, beta, depth + 1)
            if v2 < v:
                v, move = v2, a
                beta = min(beta, v)
            if v <= alpha:
                return v, move
        return v, move

    return max_value(state, -infinity, +infinity, 0)
```

Board Class

We represent states as a `Board`, which is a subclass of `defaultdict` that in general will consist of `{(x, y): contents}` pairs. A board also has some attributes:

- `.to_move` to name the player whose move it is;
- `.width` and `.height` to give the size of the board (both 3 in tic-tac-toe, but other numbers in related games);
- possibly other attributes, as specified by keywords.

As a `defaultdict`, the `Board` class has a `__missing__` method, which returns `empty` for squares that have no been assigned but are within the `width` × `height` boundaries, or `off` otherwise. The class has a `__hash__` method, so instances can be stored in hash tables.

In [73]:

```python
class Board(defaultdict):
    """A board has the player to move, a cached utility value,
    and a dict of {(x, y): player} entries, where player is 'X' or 'O'."""
    empty = '.'
    off = '#'

    def __init__(self, width=8, height=8, to_move=None, **kwds):
        self.__dict__.update(width=width, height=height, to_move=to_move, **kwds)

    def new(self, changes: dict, **kwds) -> 'Board':
        "Given a dict of {(x, y): contents} changes, return a new Board with the changes."
        board = Board(width=self.width, height=self.height, **kwds)
        board.update(self)
        board.update(changes)
        return board

    def __missing__(self, loc):
        x, y = loc
        if 0 <= x < self.width and 0 <= y < self.height:
            return self.empty
        else:
            return self.off

    def __hash__(self):
        return hash(tuple(sorted(self.items()))) + hash(self.to_move)

    def __repr__(self):
        def row(y): return ' '.join(self[x, y] for x in range(self.width))
        return '\n'.join(map(row, range(self.height))) +  '\n'
```

Player functions

We need an interface for players. We'll represent a player as a `callable` that will be passed two arguments: `(game, state)` and will return a `move`. The function player creates a player out of a search algorithm.

In [74]:
```python
def random_player(game, state): return random.choice(list(game.actions(state)))

def player(search_algorithm):
    """A game player who uses the specified search algorithm"""
    return lambda game, state: search_algorithm(game, state)[1]

def query_player(game, state):
    """Make a move by querying standard input."""
    print("current state:")
    game.display(state)
    print("available moves: {}".format(game.actions(state)))
    print("")
    move = None
    if game.actions(state):
        move_string = input('Your move? ')
        try:
            move = eval(move_string)
        except NameError:
            move = move_string
    else:
        print('no legal moves: passing turn to next player')
    return move

def depth_limit_player(search_algorithm, d):
    """A game player who uses the specified search algorithm"""
    return lambda game, state: search_algorithm(game, state, cutoff=cutoff_depth(d))[1]
```

Connect Four

Now that we have all the ingredients to start a game, let's define one, based on the `Game` class. As first game to pratice I choose Connect Four. It is a variant of tic-tac-toe, played on a larger (7 x 6) board, and with the restriction that in any column you can only play in the lowest empty square in the column.

In [75]:
```python
class ConnectFour(Game):
    """Play Connect Four on an `height` by `width` board, needing `k` in a
       row to win. 'X' plays first against 'O'."""

    def __init__(self, height=6, width=7, k=4):
        self.k = k # k in a row
        self.squares = {(x, y) for x in range(width) for y in range(height)}
        self.initial = Board(height=height, width=width, to_move='X', utility=0)

    def actions(self, board):
        """In each column you can play only the lowest empty square in the column."""
        return {(x, y) for (x, y) in self.squares - set(board)
                if y == board.height - 1 or (x, y + 1) in board}

    def result(self, board, square):
        """Place a marker for current player on square."""
        player = board.to_move
        board = board.new({square: player}, to_move=('O' if player == 'X' else 'X'))
        win = k_in_row(board, player, square, self.k)
        board.utility = (0 if not win else +1 if player == 'X' else -1)
        return board

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

    def is_terminal(self, board):
```

```
        """A board is a terminal state if it is won or there are no empty squares."""
        return board.utility != 0 or len(self.squares) == len(board)

    def display(self, board): print(board)


def k_in_row(board, player, square, k):
    """True if player has k pieces in a line through square."""
    def in_row(x, y, dx, dy): return 0 if board[x, y] != player else 1 + in_row(x + dx, y
    return any(in_row(*square, dx, dy) + in_row(*square, -dx, -dy) - 1 >= k
               for (dx, dy) in ((0, 1), (1, 0), (1, 1), (1, -1)))
```

Random players

Let's try to run a game with players that do random moves (sort of agents with random choice)

In [76]:

```
play_game(ConnectFour(), dict(X=random_player, O=random_player), verbose=True).utility
```

```
Player X move: (3, 5)
. . . . . . .
. . . . . . .
. . . . . . .
. . . . . . .
. . . . . . .
. . . X . . .

Player O move: (2, 5)
. . . . . . .
. . . . . . .
. . . . . . .
. . . . . . .
. . . . . . .
. . O X . . .

Player X move: (3, 4)
. . . . . . .
. . . . . . .
. . . . . . .
. . . . . . .
. . . X . . .
. . O X . . .

Player O move: (5, 5)
. . . . . . .
. . . . . . .
. . . . . . .
. . . . . . .
. . . X . . .
. . O X . O .

Player X move: (3, 3)
. . . . . . .
. . . . . . .
. . . . . . .
. . . X . . .
. . . X . . .
. . O X . O .

Player O move: (3, 2)
. . . . . . .
. . . . . . .
. . . O . . .
```

```
. . . X . . .
. . . X . . .
. . O X . O .

Player X move: (5, 4)
. . . . . . .
. . . . . . .
. . . O . . .
. . . X . . .
. . . X . X .
. . O X . O .

Player O move: (5, 3)
. . . . . . .
. . . . . . .
. . . O . . .
. . . X . O .
. . . X . X .
. . O X . O .

Player X move: (0, 5)
. . . . . . .
. . . . . . .
. . . O . . .
. . . X . O .
. . . X . X .
X . O X . O .

Player O move: (5, 2)
. . . . . . .
. . . . . . .
. . . O . O .
. . . X . O .
. . . X . X .
X . O X . O .

Player X move: (5, 1)
. . . . . . .
. . . . . X .
. . . O . O .
. . . X . O .
. . . X . X .
X . O X . O .

Player O move: (3, 1)
. . . . . . .
. . . O . X .
. . . O . O .
. . . X . O .
. . . X . X .
X . O X . O .

Player X move: (1, 5)
. . . . . . .
. . . O . X .
. . . O . O .
. . . X . O .
. . . X . X .
X X O X . O .

Player O move: (1, 4)
. . . . . . .
. . . O . X .
. . . O . O .
. . . X . O .
. O . X . X .
```

```
X X O X . O .

Player X move: (4, 5)
. . . . . . .
. . . O . X .
. . . O . O .
. . . X . O .
. O . X . X .
X X O X X O .

Player O move: (6, 5)
. . . . . . .
. . . O . X .
. . . O . O .
. . . X . O .
. O . X . X .
X X O X X O O

Player X move: (5, 0)
. . . . . X .
. . . O . X .
. . . O . O .
. . . X . O .
. O . X . X .
X X O X X O O

Player O move: (2, 4)
. . . . . X .
. . . O . X .
. . . O . O .
. . . X . O .
. O O X . X .
X X O X X O O

Player X move: (6, 4)
. . . . . X .
. . . O . X .
. . . O . O .
. . . X . O .
. O O X . X X
X X O X X O O

Player O move: (1, 3)
. . . . . X .
. . . O . X .
. . . O . O .
. O . X . O .
. O O X . X X
X X O X X O O

Player X move: (4, 4)
. . . . . X .
. . . O . X .
. . . O . O .
. O . X . O .
. O O X X X X
X X O X X O O
```

Out[76]:

```
1
```

The result of the game is casual, as expected. Actions are choosed without computational times.

Alphabeta with depth limit players

Now we try to instantiate two agents that uses the alphabeta search algorithms with a depth limit. The default depth is 6 moves ahead. The score of a state is just the winning, tie or loosing condition (1, 0, -1).

In [77]:
```
%time play_game(ConnectFour(), dict(X=player(h_alphabeta_search), O=player(h_alphabeta_sea
```

```
Player X move: (5, 5)
. . . . . . .
. . . . . . .
. . . . . . .
. . . . . . .
. . . . . . .
. . . . . X .

Player O move: (6, 5)
. . . . . . .
. . . . . . .
. . . . . . .
. . . . . . .
. . . . . . .
. . . . . X O

Player X move: (1, 5)
. . . . . . .
. . . . . . .
. . . . . . .
. . . . . . .
. . . . . . .
. X . . . X O

Player O move: (5, 4)
. . . . . . .
. . . . . . .
. . . . . . .
. . . . . . .
. . . . . O .
. X . . . X O

Player X move: (6, 4)
. . . . . . .
. . . . . . .
. . . . . . .
. . . . . . .
. . . . . O X
. X . . . X O

Player O move: (1, 4)
. . . . . . .
. . . . . . .
. . . . . . .
. . . . . . .
. O . . . O X
. X . . . X O

Player X move: (4, 5)
. . . . . . .
. . . . . . .
. . . . . . .
. . . . . . .
. O . . . O X
. X . . X X O

Player O move: (4, 4)
. . . . . . .
. . . . . . .
```

```
. . . . . . .
. . . . . . .
. O . . O O X
. X . . X X O

Player X move: (4, 3)
. . . . . . .
. . . . . . .
. . . . . . .
. . . . X . .
. O . . O O X
. X . . X X O

Player O move: (4, 2)
. . . . . . .
. . . . . . .
. . . . O . .
. . . . X . .
. O . . O O X
. X . . X X O

Player X move: (4, 1)
. . . . . . .
. . . . X . .
. . . . O . .
. . . . X . .
. O . . O O X
. X . . X X O

Player O move: (4, 0)
. . . . O . .
. . . . X . .
. . . . O . .
. . . . X . .
. O . . O O X
. X . . X X O

Player X move: (6, 3)
. . . . O . .
. . . . X . .
. . . . O . .
. . . . X . X
. O . . O O X
. X . . X X O

Player O move: (6, 2)
. . . . O . .
. . . . X . .
. . . . O . O
. . . . X . X
. O . . O O X
. X . . X X O

Player X move: (6, 1)
. . . . O . .
. . . . X . X
. . . . O . O
. . . . X . X
. O . . O O X
. X . . X X O

Player O move: (0, 5)
. . . . O . .
. . . . X . X
. . . . O . O
. . . . X . X
```

```
. O . . O O X
O X . . X X O

Player X move: (0, 4)
. . . . O . .
. . . . X . X
. . . . O . O
. . . . X . X
X O . . O O X
O X . . X X O

Player O move: (0, 3)
. . . . O . .
. . . . X . X
. . . . O . O
O . . . X . X
X O . . O O X
O X . . X X O

Player X move: (0, 2)
. . . . O . .
. . . . X . X
X . . . O . O
O . . . X . X
X O . . O O X
O X . . X X O

Player O move: (0, 1)
. . . . O . .
O . . . X . X
X . . . O . O
O . . . X . X
X O . . O O X
O X . . X X O

Player X move: (0, 0)
X . . . O . .
O . . . X . X
X . . . O . O
O . . . X . X
X O . . O O X
O X . . X X O

Player O move: (6, 0)
X . . . O . O
O . . . X . X
X . . . O . O
O . . . X . X
X O . . O O X
O X . . X X O

Player X move: (5, 3)
X . . . O . O
O . . . X . X
X . . . O . O
O . . . X X X
X O . . O O X
O X . . X X O

Player O move: (5, 2)
X . . . O . O
O . . . X . X
X . . . O O O
O . . . X X X
X O . . O O X
O X . . X X O
```

```
Player X move: (2, 5)
X . . . O . O
O . . . X . X
X . . . O O O
O . . . X X X
X O . . O O X
O X X . X X O

Player O move: (3, 5)
X . . . O . O
O . . . X . X
X . . . O O O
O . . . X X X
X O . . O O X
O X X O X X O

Player X move: (5, 1)
X . . . O . O
O . . . X X X
X . . . O O O
O . . . X X X
X O . . O O X
O X X O X X O

Player O move: (1, 3)
X . . . O . O
O . . . X X X
X . . . O O O
O O . . X X X
X O . . O O X
O X X O X X O

Player X move: (2, 4)
X . . . O . O
O . . . X X X
X . . . O O O
O O . . X X X
X O X . O O X
O X X O X X O

Player O move: (2, 3)
X . . . O . O
O . . . X X X
X . . . O O O
O O O . X X X
X O X . O O X
O X X O X X O

Player X move: (1, 2)
X . . . O . O
O . . . X X X
X X . . O O O
O O O . X X X
X O X . O O X
O X X O X X O

Player O move: (1, 1)
X . . . O . O
O O . . X X X
X X . . O O O
O O O . X X X
X O X . O O X
O X X O X X O

Player X move: (1, 0)
```

```
X X . . O . O
O O . . X X X
X X . . O O O
O O O . X X X
X O X . O O X
O X X O X X O

Player O move: (5, 0)
X X . . O O O
O O . . X X X
X X . . O O O
O O O . X X X
X O X . O O X
O X X O X X O

Player X move: (2, 2)
X X . . O O O
O O . . X X X
X X X . O O O
O O O . X X X
X O X . O O X
O X X O X X O

Player O move: (2, 1)
X X . . O O O
O O O . X X X
X X X . O O O
O O O . X X X
X O X . O O X
O X X O X X O

Player X move: (2, 0)
X X X . O O O
O O O . X X X
X X X . O O O
O O O . X X X
X O X . O O X
O X X O X X O

Player O move: (3, 4)
X X X . O O O
O O O . X X X
X X X . O O O
O O O . X X X
X O X O O O X
O X X O X X O

Player X move: (3, 3)
X X X . O O O
O O O . X X X
X X X . O O O
O O O X X X X
X O X O O O X
O X X O X X O

Wall time: 7.28 s
```

Out[77]:

```
1
```

We observe that this time there are computational times. They aren't enormous thanks to the cutoff in depth. With the same configuration, player `X` will always win.

Changing depth limit

Let's try to play with the `d` factor. It's clear that the `X` has an advantage beacouse starts first, so we limit its searching algorithm to 1 move ahead. Now we increase the depth limit of `O` player algorithm to see whe it'll win.

In [78]:
```
%time play_game(ConnectFour(), dict(X=depth_limit_player(h_alphabeta_search, 1), O=depth_l
```

```
Player X move: (5, 5)
. . . . . . .
. . . . . . .
. . . . . . .
. . . . . . .
. . . . . . .
. . . . . X .

Player O move: (6, 5)
. . . . . . .
. . . . . . .
. . . . . . .
. . . . . . .
. . . . . . .
. . . . . X O

Player X move: (1, 5)
. . . . . . .
. . . . . . .
. . . . . . .
. . . . . . .
. . . . . . .
. X . . . X O

Player O move: (5, 4)
. . . . . . .
. . . . . . .
. . . . . . .
. . . . . . .
. . . . . O .
. X . . . X O

Player X move: (6, 4)
. . . . . . .
. . . . . . .
. . . . . . .
. . . . . . .
. . . . . O X
. X . . . X O

Player O move: (1, 4)
. . . . . . .
. . . . . . .
. . . . . . .
. . . . . . .
. O . . . O X
. X . . . X O

Player X move: (4, 5)
. . . . . . .
. . . . . . .
. . . . . . .
. . . . . . .
. O . . . O X
. X . . X X O

Player O move: (4, 4)
. . . . . . .
```

```
. . . . . . .
. . . . . . .
. . . . . . .
. O . . O O X
. X . . X X O

Player X move: (4, 3)
. . . . . . .
. . . . . . .
. . . . . . .
. . . . X . .
. O . . O O X
. X . . X X O

Player O move: (4, 2)
. . . . . . .
. . . . . . .
. . . . O . .
. . . . X . .
. O . . O O X
. X . . X X O

Player X move: (4, 1)
. . . . . . .
. . . . X . .
. . . . O . .
. . . . X . .
. O . . O O X
. X . . X X O

Player O move: (4, 0)
. . . . O . .
. . . . X . .
. . . . O . .
. . . . X . .
. O . . O O X
. X . . X X O

Player X move: (6, 3)
. . . . O . .
. . . . X . .
. . . . O . .
. . . . X . X
. O . . O O X
. X . . X X O

Player O move: (6, 2)
. . . . O . .
. . . . X . .
. . . . O . O
. . . . X . X
. O . . O O X
. X . . X X O

Player X move: (6, 1)
. . . . O . .
. . . . X . X
. . . . O . O
. . . . X . X
. O . . O O X
. X . . X X O

Player O move: (0, 5)
. . . . O . .
. . . . X . X
. . . . O . O
```

```
. . . . X . X
. O . . O O X
O X . . X X O

Player X move: (0, 4)
. . . . O . .
. . . . X . X
. . . . O . O
. . . . X . X
X O . . O O X
O X . . X X O

Player O move: (0, 3)
. . . . O . .
. . . . X . X
. . . . O . O
O . . . X . X
X O . . O O X
O X . . X X O

Player X move: (0, 2)
. . . . O . .
. . . . X . X
X . . . O . O
O . . . X . X
X O . . O O X
O X . . X X O

Player O move: (0, 1)
. . . . O . .
O . . . X . X
X . . . O . O
O . . . X . X
X O . . O O X
O X . . X X O

Player X move: (0, 0)
X . . . O . .
O . . . X . X
X . . . O . O
O . . . X . X
X O . . O O X
O X . . X X O

Player O move: (6, 0)
X . . . O . O
O . . . X . X
X . . . O . O
O . . . X . X
X O . . O O X
O X . . X X O

Player X move: (5, 3)
X . . . O . O
O . . . X . X
X . . . O . O
O . . . X X X
X O . . O O X
O X . . X X O

Player O move: (5, 2)
X . . . O . O
O . . . X . X
X . . . O O O
O . . . X X X
X O . . O O X
```

```
O X . . X X O

Player X move: (2, 5)
X . . . O . O
O . . . X . X
X . . . O O O
O . . . X X X
X O . . O O X
O X X . X X O

Player O move: (3, 5)
X . . . O . O
O . . . X . X
X . . . O O O
O . . . X X X
X O . . O O X
O X X O X X O

Player X move: (5, 1)
X . . . O . O
O . . . X X X
X . . . O O O
O . . . X X X
X O . . O O X
O X X O X X O

Player O move: (1, 3)
X . . . O . O
O . . . X X X
X . . . O O O
O O . . X X X
X O . . O O X
O X X O X X O

Player X move: (2, 4)
X . . . O . O
O . . . X X X
X . . . O O O
O O . . X X X
X O X . O O X
O X X O X X O

Player O move: (1, 2)
X . . . O . O
O . . . X X X
X O . . O O O
O O . . X X X
X O X . O O X
O X X O X X O

Player X move: (1, 1)
X . . . O . O
O X . . X X X
X O . . O O O
O O . . X X X
X O X . O O X
O X X O X X O

Player O move: (2, 3)
X . . . O . O
O X . . X X X
X O . . O O O
O O O . X X X
X O X . O O X
O X X O X X O
```

```
Player X move: (1, 0)
X X . . O . O
O X . . X X X
X O . . O O O
O O O . X X X
X O X . O O X
O X X O X X O

Player O move: (5, 0)
X X . . O O O
O X . . X X X
X O . . O O O
O O O . X X X
X O X . O O X
O X X O X X O

Player X move: (3, 4)
X X . . O O O
O X . . X X X
X O . . O O O
O O O . X X X
X O X X O O X
O X X O X X O

Player O move: (3, 3)
X X . . O O O
O X . . X X X
X O . . O O O
O O O O X X X
X O X X O O X
O X X O X X O

Wall time: 12.9 s
```

Out[78]:
```
-1
```

Finally, with a depth limit of 8, `O` wins against a first move 1 move ahead `X` player. That' said, it's also true that the computation becomes less time efficient.

Conclusions

We observe that putting a depth limit in the computation can make the finding of an action faster, but reduces effectiveness. In fact player `O` beats an `X` player with depth limit **1**, which starts first in the game, only with a depth limit of **8**. \ That's becouse in this test we used a poor utility function (it just finds a winning solution in the future), but with a properly heuristic we can find a compromise between performance and computation time.

Next steps

This notebook is just a study of the material provided by the **"Artificial Intelligence - A Modern Approach"** book by Russell and Norvig. In the next weeks I'll try to implement a chess game with this logic, using agents with different searching algorithms and heuristics.

In [ ]: