Assignment 1

Håkon Mork
ESCE 526 Artificial Intelligence

February 1, 2013

1

1.1 Number of states visited

Game A:

Cutoff depth	3	4	5	6
Minmax	77,445	1,276,689		
α-β pruning	4,129	48,203		
Improvement	×18.76	×26.49		

Game B:

Cutoff depth	3	4	5	6
Minmax				
α-β pruning				
Improvement				

Game C:

Cutoff depth	3	4	5	6
Minmax				
α-β pruning				
Improvement				

1.2 Does state generation order matter?

My evaluation function iterates through the successor states in the order they were generated: left-to-right, top-to-bottom, with the directions generated in the (arbitrary) order north-east-south-west. I shuffled[1] the list of possible successors before evaluating their subtrees, which should indicate whether there were any serious discrepancies in the number of states visited.

1.3 Delaying defeat

2

2.1 Choice of evaluation function

2.2 Number of nodes visited

2.3 Tradeoff between evaluation function and game tree depth

[1] I used the Python standard library's `random.shuffle` function, which shuffles a sequence (e.g., our list of successor states) in place.

A Appendix: Source code

A.1 Implementation comments

The code was written in Python, which is one of the languages I am most familiar with.

I consider the successors of the current state to be on cutoff level 0, and pass a depth parameter to the alpha-beta and minmax algorithms. The depth parameter is incremented for each ply in the recursion tree until the given cutoff limit is hit, at which point the heuristic evaluation of that state is returned.

A.2 Listing

```python
#!/usr/bin/env python

import string, copy, time, logging

# debug < info < warning < error < critical?
logging.basicConfig(level=logging.CRITICAL)

withhuman = False # human vs. computer, or computer against itself
logthegame = False # write a log file on exit
useab = True # alphabeta or minmax
fancy = False # simple or fancy heuristic

statesvisited = 0

# we store the board as a matrix, i.e., a list of lists
# initialstate = [
#   ["O", None, None, None, None, None, "X"],
#   ["X", None, None, None, None, None, "O"],
#   ["O", None, None, None, None, None, "X"],
#   ["X", None, None, None, None, None, "O"],
#   ["O", None, None, None, None, None, "X"],
#   ["X", None, None, None, None, None, "O"],
#   ["O", None, None, None, None, None, "X"]
# ]

class Node:
    def __init__(self, board, player, command):
        self.board = board
        self.player = player
        self.value = fancyheuristic(board, player) if fancy else simpleheuristic(board, player)
        self.command = command # the move made to generate this state

class Black:
    def __init__(self):
        self.piece = "X"

class White:
    def __init__(self):
        self.piece = "O"

def successors(board, player):
    logging.debug("Generating successors for player = " + player.__class__.__name__ + ", board = " + str(board))
    succs = []
    for y, line in enumerate(board):
        for x, char in enumerate(line):
            if char == player.piece:
                # try all possible moves: xyN, xyE, xyS, xyW
                for cmd in (str(x + 1) + str(y + 1) + d for d in directions):
                    #print player.__class__.__name__, cmd,
                    try:
                        candidate = move(cmd, board, player)
                        succs.append(Node(candidate, player, cmd))
                        #print "works ->", len(succs)
                    except (ValueError, IndexError) as e:
                        # ValueError: attempted move was illegal, e.g. trying to move to an occupied square
                        # IndexError: try to move outside of the board
                        #print "".join(e)
                        continue
    logging.debug("There were " + str(len(succs)) + " successors")
    return succs

def alphabeta(player, node, depth, alpha, beta):
    global statesvisited
    statesvisited += 1
    succs = successors(node.board, player)
    otherplayer = black if player is white else black
    logging.info("Inside alphabeta on node " + str(hash(node)) + " obtained by " + node.command)
    logging.info(str(hash(node)) + " looks like\n" + prettyprint(node.board))
    logging.info(str(hash(node)) + " has depth = " + str(depth) + ", children = " + str(len(succs)))
    logging.debug("They are (" + player.__class__.__name__ + "): ")
    logging.debug("\n".join([c.command + " -> node " + str(hash(c)) for c in succs]))
    if depth == cutoff or len(succs) == 0:
        logging.info("Bottom reached, return utility " + str(node.value) + " from " + str(hash(node)))
        if node.value > 0:
            logging.info("Win found:\n" + prettyprint(node.board))
        return node.value
    elif player is white: #maxplayer, arbitrary
        logging.debug("State is \n" + prettyprint(node.board))
        for childnode in succs:
            logging.debug("Entering examination of child " + str(hash(childnode)) + " by " + childnode.command + " from " + str(hash(node)))
            alpha = max(alpha, alphabeta(otherplayer, childnode, depth + 1, alpha, beta))
            if alpha >= beta:
                logging.info("Pruning: returning beta = " + str(beta) + " from " + str(hash(childnode)))
                return beta
        logging.info("No pruning: returning alpha = " + str(alpha) + " from " + str(hash(node)))
        return alpha
    else: #black minplayer
        logging.debug("State is \n" + prettyprint(node.board))
        for childnode in succs:
            logging.debug("Entering examination of child " + str(hash(childnode)) + " by " + childnode.command + " from " + str(hash(node)))
            beta = min(beta, alphabeta(otherplayer, childnode, depth + 1, alpha, beta))
            if alpha >= beta:
                logging.info("Pruning: returning alpha = " + str(alpha) + " from " + str(hash(childnode)))
                return alpha
        logging.info("No pruning: returning beta = " + str(beta) + " from " + str(hash(node)))
        return beta

def minmax(player, node, depth):
    global statesvisited
    statesvisited += 1
    logging.debug("Inside minmax on node " + str(hash(node)) + " depth = " + str(depth))
    #otherplayer = white if player is black else black
    minplayer = black # arbitrary
    if depth == cutoff or not successors(node.board, player):
        logging.debug("Bottom reached, return utility " + str(node.value))
        if node.value > 0:
            logging.debug("Win found:\n" + prettyprint(node.board))
        return node.value
    elif node.player is minplayer:
        logging.debug("Recursive minmax: player " + str(player) + ", depth = " + str(depth) + ", node = " + str(hash(node)))
        return min(minmax(player, child, depth + 1) for child in successors(node.board, player))
    else:
        logging.debug("Recursive minmax: player " + str(player) + ", depth = " + str(depth) + ", node = " + str(hash(node)))
        return max(minmax(player, child, depth + 1) for child in successors(node.board, player))

def prettyprint(board):
    b = "\n".join(",".join(map(str, row)) for row in board)
    return b.replace("None", " ")

def winner(board):
    # indicate the winner (if any) in the given board state
    def horizontal(board):
        # check if any consecutive four entries in a row are X-es or O-s
        for line in board:
            for i, char in enumerate(line):
                if line[i : i + 4] == ["O"] * 4:
                    return white
                elif line[i : i + 4] == ["X"] * 4:
                    return black
    def vertical(board):
        # equivalent to the horizontal winner in the transposed matrix
        return horizontal(map(list, zip(*board)))
    def diagonal(board):
        # all downward diagonals must start in the upper-left 4x4 submatrix
        # similarly, all upward diagonals must start in the lower-left 4x4 submatrix
        # somewhat inelegant, but it works
        for i in range(4):
            for j in range(4):
                if all(board[i + k][j + k] == "O" for k in range(4)) or all(board[6 - i - k][j + k] == "O" for k in range(4)):
                    return white
                elif all(board[i + k][j + k] == "X" for k in range(4)) or all(board[6 - i - k][j + k] == "X" for k in range(4)):
                    return black
    return horizontal(board) or vertical(board) or diagonal(board)

def simpleheuristic(board, player):
    otherplayer = white if player is black else black
    if winner(board) is player:
        return 1
    elif winner(board) is otherplayer:
        return -1
    else:
        return 0

def fancyheuristic(board, player):
    pass

def parse(boardstring):
    # build a matrix from a string describing the board layout
    boardstring = string.replace(boardstring, ",", "")
    board, line = [], []
    for char in boardstring:
        if char == " ":
            line.append(None)
        elif char == "\n":
            board.append(line)
            line = []
        else:
            line.append(char)
    if line:
        board.append(line) # last line, if there is no newline at the end
    return board


def move(command, board, player):
    # takes indices and a direction, e.g. "43W" or "26N"
    x, y, d = tuple(command)
    # the board is a zero-indexed array, adjust accordingly
    x, y = int(x) - 1, int(y) - 1
    dy, dx = directions[d.upper()]
    # does the piece fall within the bounds?
    if ((0 <= x + dx <= 7) and (0 <= y + dy <= 7)
    # and is it our piece?
    and board[y][x] == player.piece
    # and is the destination square empty?
    and not board[y + dy][x + dx]):
        # then it's okay
        # we don't want to update in place
        successor = copy.deepcopy(board)
        successor[y + dy][x + dx] = successor[y][x]
        successor[y][x] = None
        return successor
    else:
        raise ValueError#("The move " + command + " is not legal")

white = White()
black = Black()
human = white if withhuman else None
computer = black
currentplayer = white
cutoff = 5

# tuples of (dy, dx) for all directions
directions = {
    "N": (-1, 0),
    "E": (0, 1),
    "S": (1, 0),
    "W": (0, -1)
}
```

```

with open("./starta.txt", "r") as f:
    initstatestr = f.read()
board = parse(initstatestr)

#board = initialstate
log = ["Initial state:"]
movenumber = 1

while winner(board) is None:
    playername = currentplayer.__class__.__name__
    p = prettyprint(board)
    print p
    log.append(p)
    print "\nMove #%s:" % movenumber
    log.append("\nMove #%s:" % movenumber)
    cmd = ""
    print "It's %s's turn." % playername
    try:
        if currentplayer is human:
            print "Possible moves:"
            for s in successors(board, currentplayer):
                print s.command
            cmd = raw_input()
        else: #let the computer play against itself
            succs = successors(board, currentplayer)
            # take the possible move now, pick something better later on if we can find it
            bestmove = succs[0].command
            bestutility = 0
            if useab: #alphabeta
                logging.warning("Player " + playername + " thinking about what to do.")
                logging.warning("Using alphabeta with cutoff " + str(cutoff))
                for succboard in succs:
                    #init with alpha = -inf, beta = inf
                    u = alphabeta(currentplayer, succboard, 0, float("-inf"), float("inf"))
                    if u > bestutility:
                        bestutility = u
                        bestmove = succboard.command
```

```python
            else: #minmax
                logging.warning("Player " + playername + " thinking about what to do.")
                logging.warning("Using minmax with cutoff " + str(cutoff))
                for succboard in succs:
                    u = minmax(currentplayer, succboard, 0)
                    if u > bestutility:
                        logging.critical("Utility improved: " + str(u) + " from " + succboard.command)
                        bestutility = u
                        bestmove = succboard.command
            cmd = bestmove
            print "The computer makes the move", cmd

        print "cutoff", cutoff, "states", statesvisited, "with", "alphabeta" if useab else "minmax"
        raise Exception("Counting states visited")
        board = move(cmd, board, currentplayer)
        log.append("%s plays %s." % (playername, cmd))
        currentplayer = white if currentplayer is black else black
        playername = currentplayer.__class__.__name__
        movenumber += 1
    #except ValueError:
    #   print "Illegal move."
        #raise
    except KeyboardInterrupt:
        log.append("Game cancelled.")
        logging.critical("Game cancelled.")
        break

# post-game cleanup
print prettyprint(board)
log.append(prettyprint(board))

if winner(board):
    s = "%s won the match" % winner(board).__class__.__name__
    print s
    log.append(s)
else:
    print "It's a draw"
    log.append("It's a draw")
```

```python

if logthegame:
    logname = time.strftime("/Users/hakon/Desktop/con4-%Hh%M-%S.log")
    with open(logname, "w+") as logfile:
        logfile.write("\n".join(log))
```