Assignment 1

Håkon Mork
ESCE 526 Artificial Intelligence

February 3, 2013

1

1.1 Number of states visited with simple heuristic

Game A:

Cutoff depth	3	4	5	6
Minmax	77,445	1,276,689	21,335,620	
α-β pruning	4129	48,203	694,652	
Improvement	×18.76	×26.49	×30.72	?

Game B:

Cutoff depth	3	4	5	6
Minmax	98,345	1,704,319	29,770,996	
α-β pruning	6421	96,884	1,683,194	
Improvement	×15.32	×17.59	×17.69	?

Game C:

Cutoff depth	3	4	5	6
Minmax	69,954	1,237,535	22,191,032	
α-β pruning	3763	51,098	840,633	
Improvement	×18.59	×24.22	×26.40	?

In my code I consider the children of the current state to be on cutoff level 0; their children, i.e., the grandchildren of the current state, are on level 1, and so on.[1] The heuristic evaluation of the node is returned when the depth reaches the limit given by the `--cutoff` argument.

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 considered the first move in game A and ran[2] alpha-beta pruning five times with a cutoff depth of 3, shuffling the list of successor states randomly every time one is generated. (Since the minmax algorithm does not prune the game tree at all, the order in which it evaluates successors is irrelevant.) Alpha-beta pruning with the non-shuffled evaluation order visited 4129 states, as per the table above.

I found that evaluation order *did* matter, though not impressively so. The sample runs visited 4480, 4338, 4324, 4114, and 4376 states, respectively, for an average of 4326 states, which is 4.77% more than the non-shuffled case. The maximum deviations were 8.5% more and 0.36% fewer states visited than the original order. I also tried evaluating states in the reverse order; the difference was negligble. This suggests that there is no significant benefit to be gained by shuffling; in fact, we see that more states were visited with alternative evaluation orders than with the original sequence. I don't know if this is a coincidence or if the order I picked is somehow optimal.

2

2.1 Choice of evaluation function

$$s = \sum_{i=2}^{4} n_i \cdot 10^i$$

[1] If the current state's children should instead be regarded as being on level 1, the values in the tables above should be shifted one column to the right.

[2] The commands given were `python ass1.py --input starta.txt --cutoff 3 --alg ab --count --shuffle`. See the appendix for details on usage.

2.2 Number of states visited with advanced heuristic

2.3 Tradeoff between evaluation function and game tree depth

My evaluation function is not particularly efficient. It does

A Appendix: Source code

A.1 Usage

All arguments are optional:

- `-i` or `--input`: Specify an input file to be used as the initial game state. A plain-text file following the notation used in the assignment is expected. Defaults to the example illustrated in the “Introduction” part of the assignment text.
- `-u` or `--human`: The computer should play against a human adversary, not just against itself. The user will be prompted for input when it is their turn to play.
- `-c` or `--cutoff`: Specify a cutoff depth. Defaults to 3.
- `-a` or `--alg`: Specify which of the minmax or alpha-beta pruning algorithms is to be used. May take values `mm` or `ab`. Defaults to alpha-beta pruning.
- `-l` or `--log`: A log file should be written on exit.
- `-k` or `--count`: Count the number of states visited. Used for problem 1.1.
- `-s` or `--shuffle`: Shuffle the list of successor states before evaluating them. Used for problem 1.2.

Example: `python ass1.py --input file.txt --alg ab --human --log`

A.2 Listing

I’ve expunged all logging statements and other debugging aids for the sake of readability.

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

import string, copy, time, logging, argparse, random

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

#withhuman = False # human vs. computer, or computer against itself
#fancy = False # simple or fancy heuristic

# tuples of (dy, dx) for all directions
```

```python
directions = {
    "N": (-1, 0),
    "E": (0, 1),
    "S": (1, 0),
    "W": (0, -1)
}

# used for counting states, problem 1.1
statesvisited = 0

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):
                    try:
                        candidate = move(cmd, board, player)
                        succs.append(Node(candidate, player, cmd))
                    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
                        continue
    logging.debug("There were " + str(len(succs)) + " successors")
    if args.shuffle:
        random.shuffle(succs)
    return succs

def alphabeta(player, node, depth, alpha, beta):
    if countingstates:
        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]))
    # cut off if we are too deep down
    if depth == cutoff or len(succs) == 0:
        logging.info("Bottom reached, return utility " + str(node.value) + " from " + str(hash(node)))
        return node.value
    # return immediately if we win by making this move
    elif winner(node.board) is player:
        return float("inf")
    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):
    if countingstates:
        global statesvisited
        statesvisited += 1
    logging.debug("Inside minmax on node " + str(hash(node)) + " depth = " + str(depth))
    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))
```

```python

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

def horizontal(board, n):
    # check if any consecutive n entries in a row are X-es or O-s
    for line in board:
        for i, char in enumerate(line):
            if line[i : i + n] == ["O"] * n:
                return white
            elif line[i : i + n] == ["X"] * n:
                return black

def vertical(board, n):
    # equivalent to the horizontal winner in the transposed matrix
    return horizontal(map(list, zip(*board)), n)

def diagonal(board, n):
    # 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(n):
        for j in range(n):
            if all(board[i + k][j + k] == "O" for k in range(n)) or all(board[6 - i - k][j + k] == "O" for k in range(n)):
                return white
            elif all(board[i + k][j + k] == "X" for k in range(n)) or all(board[6 - i - k][j + k] == "X" for k in range(n)):
                return black

def winner(board):
    # indicate the winner (if any) in the given board state
    return horizontal(board, 4) or vertical(board, 4) or diagonal(board, 4)

def closeness(board, player):
    pass
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):
    otherplayer = white if player is black else white
    score = 0
    for i in [4, 3, 2]:
        n = 0
        if horizontal(board, i) is player or vertical(board, i) is player or diagonal(board, i) is player:
            n += 1
        if horizontal(board, i) is otherplayer or vertical(board, i) is otherplayer or diagonal(board, i) is otherplayer:
            n -= 1
        score += (10 ** i) * n
#   score = 10 ** inarow(board, player) - 0.5 * 10 ** inarow(board, otherplayer)
    #if
    return score

def parseboard(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")


parser = argparse.ArgumentParser()
parser.add_argument("-c", "--cutoff", help="Cutoff depth")
parser.add_argument("-i", "--input", help="Input game board")
parser.add_argument("-u", "--human", help="Play with a human opponent", action="store_true")
parser.add_argument("-a", "--alg", choices=["mm", "ab"], help="Minmax or alpha-beta algorithm")
parser.add_argument("-l", "--log", help="Write a game log on exit", action="store_true")
parser.add_argument("-s", "--shuffle", help="Shuffle successor list", action="store_true")
parser.add_argument("-k", "--count", help="Count states visited", action="store_true")
parser.add_argument("-f", "--fancy", help="Fancy heuristic function", action="store_true")
args = parser.parse_args()

cutoff = int(args.cutoff) if args.cutoff else 3
useab = not (args.alg == "mm")
logthegame = args.log
```

```
countingstates = args.count
fancy = args.fancy

if args.input:
    with open(args.input, ”r”) as inputfile:
        initstr = inputfile.read()
    board = parseboard(initstr)
else:
    board = [
        [”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”]
    ]

white = White()
black = Black()
human = white if args.human else None
computer = black
currentplayer = white

log = [”Initial state:”]
movenumber = 1

t = time.time()

while winner(board) is None:
    playername = currentplayer.__class__.__name__
    p = prettyprint(board)
    print p
    print ”\nMove #%s:” % movenumber
    print ”It’s %s’s turn.” % playername
    if logthegame:
        log.append(p)
        log.append(”\nMove #%s:” % movenumber)
        log.append(”It’s %s’s turn.” % playername)
    cmd = ””
    try:
```

```python
        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 first move, 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
            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 "Thinking took", time.time() - t, "seconds"
            if logging:
                log.append("Thinking took " + str(time.time() - t) + " seconds")

        board = move(cmd, board, currentplayer)

        if countingstates:
            print statesvisited
            raise Exception("Counting states, stopping here")
        if logthegame:
            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:
        if logthegame:
            log.append("Game cancelled.")
        logging.critical("Game cancelled.")
        break

# post-game cleanup
print prettyprint(board)

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

if logthegame:
    log.append(prettyprint(board))
    logname = time.strftime("./connect4-%H-%M-%S.log")
    with open(logname, "w+") as logfile:
        logfile.write("\n".join(log))
```