Assignment 1

Håkon Mork
ESCE 526 Artificial Intelligence

February 4, 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	Unknown
α-β pruning	4129	48,203	694,652	Unknown
Improvement	$\times$18.76	$\times$26.49	$\times$30.72	Unknown

Game B:

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

Game C:

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

This game has a high branching factor, in the ballpark of about 16 possible moves for each round, which makes exploring the game tree to any significant depth a very time-consuming chore. I didn't have time to calculate the numbers for cutoff depth 6, but they should be at least one order of magnitude higher than for the previous depth. Depressingly, I also found a bug in my program shortly before the submission deadline, so I'm worried that I have miscalculated the minmax figures. An interesting observation is that alpha-beta pruning seems to be increasingly efficient at chopping off irrelevant branches of the game tree as the cutoff depth increases.[1] Observe, for example, that in game A the number of states visited by minmax with cutoff depth 3 was 18 times higher than the number visited by alpha-beta, while it was over 30 times higher when the cutoff depth was 5.

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.)

I found that evaluation order *did* matter, though not impressively so. Alpha-beta pruning with the non-shuffled evaluation order visited 4129 states, as per the table above. The sample runs with suffling 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 from the original order were 8.5% more and 0.36% fewer states visited. (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; alpha-beta pruning should perform best when the most valuable branches of the game tree are explored early, and most of the subsequent branches are pruned.

2

2.1 Choice of evaluation function

The `fancyheuristic` function calculates a heuristic value for a given player in a given board state. The heuristic has two parts: one to give a score based on how good the player's position is, and one to give a score for foiling the other player's plans. I underestimated the time it would take to implement a good heuristic, and due to coursework in other subject I didn't get time to work on it as much as I would have liked.

The part of the heuristic that evaluates the value of the board positions is rather simple. Let n_i be the number of i-in-a-row instances the player has on the board, either horizontally, vertically, or diagonally. For example, if the player has 3 pieces in a row at two different spots on the board, we have $n_3 = 2$. This number is multiplied by a corresponding power of 10, so that more weight is given to board states with more pieces connected. That is, the score is $\sum_{i=2}^{4} n_i \cdot 10^i$. This ensures that the player gets a higher score for having more pieces in a row; that is, having several instances of two pieces connected is by far outweighed by having three pieces connected. Though this is hardly an optimal weighting, from my observations it looks like it performs reasonably well.

[1] In my code I consider the children of the current state to be in ply 0; their children, i.e., the grandchildren of the current state, are in ply 1, and so on. If this interpretation is incorrect and the current state's children should instead be regarded as being on ply 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.

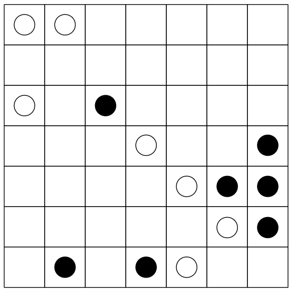

Figure 1: Example position.

The heuristic also comprises the `sabotage` function, which gives bonus points for obstructing the opponent. If the opponent has three pieces in a row and is just about to win, placing a piece such that an end of that line is blocked confers a big bonus on that game state's heuristic value. The rationale is that only one thing should be better than hindering the opponent from connecting four pieces: namely, to connect four of one's own pieces. Therefore the penalty or bonus is 9999, which is barely smaller than the 10,000 gained from winning. I haven't been able to say conclusively whether the agent plays better with this addition, but it stands to reason that it should.

As an example of how the heuristic works, consider the situation in figure 1, where the white player gets $1 \times 10^3 + 2 \times 10^2 = 1200$ points for having that many pieces in a row. The black player gets a bonus for blocking a white connect-4 while collecting $1 \times 10^3 + 1 \times 100 = 1100$ points for the rest of the pieces.

In a sense, my agent follows a greedy algorithm: it follows a (locally) utility-maximizing sequence of actions. Thus, since winning the game has the highest possible heuristic value, the agent will try to achieve that outcome as soon as it can, without any specific regard to the inevitability (or not) of that outcome. Likewise, since losing has low utility but blocking the opponent from winning has high utility, the agent will try to prevent the opponent's victory. It does not specifically try to delay its own defeat, other than in terms of choosing the course of action that maximizes utility. Hopefully those concerns will coincide.

2.2 Number of states visited with advanced heuristic

The improved heuristic does *not* consistently reduce the number of states visited, which I found surprising: for example, with alpha-beta pruning and a cutoff depth of 3, it visited 11,152 states, compared to the simple heuristic's 4129 in the first move of game A. With a cutoff depth of 4 it visited 49,254, which is comparable to the simple heuristic's 48,203. With minmax it visits the same number of states since the game tree isn't pruned. While it may be the case that my improved heuristic just isn't any good, it could also be that a significant number of states have the same heuristic value. I find it hard to believe that there should be more of those than all the states with value 0 in the case of the simple heuristic, so I have no plausible explanation.

2.3 Tradeoff between evaluation function and game tree depth

The improved heuristic is significantly more computationally intensive, and takes longer to explore the game tree to a given depth than the simple heuristic. An agent using the simple one can thus explore a greater number of states within a given amount of time. While this is certainly a drawback for the advanced heuristic, I found that the agent's gameplay improved significantly, even when severely constrained by time or cutoff depth. In contrast, the simple heuristic, even when given more time and a greater depth, saw my agent explore a vastly greater game tree and still come up with obviously boneheaded decisions. Thus it seems to me that improving the quality of the heuristic is preferable to throwing more hardware at a simple, stupid heuristic to allow it to look even further ahead and explore the state space even more. That is, as long as we can't have our cake and eat it too.

A Appendix: Source code

A.1 Implementation comments

The implementation is single-threaded, which is obviously not optimal in terms of pure performance. It seems to me that exploring a state space is a problem that lends itself well to parallelization since the subproblems are independent and potentially non-overlapping.

A.2 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. May take values `w` or `b` to indicate that the human should be white or black, respectively.
- `-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. May prove useful for the tournament.
- `-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.
- `-f` or `--fancy`: Indicate that the advanced heuristic should be used.
- `-t` or `--time`: Time limit (in seconds) for each move.

Example: `python ass1.py --input file.txt --alg ab --human w --time 20 --fancy --log`

A.3 Listing

The code is written in Python 2.7. I've expunged all logging statements and other debugging aids from this listing for the sake of readability. I find that Python is quite legible in most cases, so I've primarily added comments to explain purpose.

```
#!/usr/bin/env python
# -*- coding: utf-8 -*-
# Usage: python ass1.py [args]
# See submitted report for details.

import string, copy, time, argparse, random
```

```python

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

# Used for counting states, problem 1.1
statesvisited = 0

# Used for the search algorithms.
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

# Dummy classes for representing the players.
class Black:
    def __init__(self):
        self.piece = "X"

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

# Generates a list of all possible successor states to the given board
    position.
def successors(board, player):
    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)
```

```python
                            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
    # Used for problem 1.2, for determining whether varying the evaluation
        order matters
    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 = white if player is black else black
    # Cut off and return heuristic value if we are too deep down
    if depth == cutoff or len(succs) == 0:
        return node.value
    # White is maxplayer (arbitrary pick)
    elif player is white:
        for childnode in succs:
            alpha = max(alpha, alphabeta(otherplayer, childnode, depth + 1,
                alpha, beta))
            if alpha >= beta:
                return beta
        return alpha
    # Black is minplayer
    else:
        for childnode in succs:
            beta = min(beta, alphabeta(otherplayer, childnode, depth + 1,
                alpha, beta))
            if alpha >= beta:
                return alpha
        return beta

def minmax(player, node, depth):
    otherplayer = white if player is black else black
    if countingstates:
```

```
        global statesvisited
        statesvisited += 1
    if depth == cutoff or not successors(node.board, player):
        return node.value
    elif node.player is black: # Arbitrary pick
        return min(minmax(otherplayer, child, depth + 1) for child in successors(node.board, player))
    else:
        return max(minmax(otherplayer, child, depth + 1) for child in successors(node.board, player))

# Returns a comma-separated board of X-es and O-s to be printed to console.
def prettyprint(board):
    b = "\n".join(",".join(map(str, row)) for row in board)
    return b.replace("None", " ")

# Check if any consecutive n entries in a row are X-es or O-s, and
# return the number of n-in-a-row instances on the board for the given player.
def horizontal(board, n, player):
    piece = player.piece
    connected = 0
    for line in board:
        for i, char in enumerate(line):
            if line[i : i + n] == [piece] * n:
                connected += 1
    return connected

# Checking verticals is equivalent to checking horizontals in the transposed matrix.
def vertical(board, n, player):
    return horizontal(map(list, zip(*board)), n, player)

# All downward diagonals must start in the upper-left 4x4 submatrix, and similarly, all upward diagonals must start in the lower-left 4x4 submatrix.
# Somewhat inelegant, but it works.
def diagonal(board, n, player):
    piece = player.piece
    connected = 0
    for i in range(n):
        for j in range(n):
```

```
            # Count four down(up)ward from the upper (lower) left quadrant.
            if (all(board[i + k][j + k] == piece for k in range(n))
            or all(board[6 - i - k][j + k] == piece for k in range(n))):
                connected += 1
    return connected

# Indicate the winner (if any) in the given board state.
# Used, among other things, for the main game loop, which runs as long as there is no winner.
def winner(board):
    if horizontal(board, 4, white) or vertical(board, 4, white) or diagonal(board, 4, white):
        return white
    elif horizontal(board, 4, black) or vertical(board, 4, black) or diagonal(board, 4, black):
        return black
    else:
        return None

# Indicated whether the player has managed to thwart the opponent's play by blocking three of their pieces, thus preventing a loss.
# Used by the advanced utility function.
def sabotage(board, player):
    goal = "OOOX" if player is black else "XXXO"
    # This is a terrible, terrible hack, and I'm ashamed of it.
    # Map the elements in the matrix to strings, concatenate,
    auxboard = [map(str, l) for l in copy.deepcopy(board)]
    auxboard = ["".join(l) for l in auxboard]
    auxtransp = ["".join(l) for l in map(list, zip(*auxboard))]
    # then look up XXXO and OOOX and their reverses in that string.
    hor = any(goal in line or goal[::-1] in line for line in auxboard)
    vert = any(goal in line or goal[::-1] in line for line in auxtransp)
    # The diagonal is a bit more tricky, but the same reasoning applies as in the horizontal(board, n, player) function.
    # All interesting diagonals start in the upper or lower left quandrants, so we make a list of them, join each of them up and look for the OOOX and XXXO strings and their reverses there.
    diags = []
    for i in range(4):
        for j in range(4):
            diags.append([board[i + k][j + k] for k in range(4)])
```

```python
            diags.append([board[6 - i - k][j + k] for k in range(4)])
    # Map elements to string and concatenate with empty string.
    diags = ["".join(l) for l in [map(str, l) for l in diags]]
    diag = any(goal in line or goal[::-1] in line for line in diags)
    return hor or vert or diag

# As given in problem 1.
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

# A somewhat more advanced heuristic, used for part 2 of the assignment and
    actual gameplay. See the submitted report for details and discussion.
def fancyheuristic(board, player):
    otherplayer = white if player is black else black
    score = 0
    for i in [4, 3, 2]:
        h = horizontal(board, i, player)
        v = vertical(board, i, player)
        d = diagonal(board, i, player)
        score += (10 ** i) * (h + v + d)
    if sabotage(board, player):
        score += 9999
    elif sabotage(board, otherplayer):
        score -= 9999
    return score

# Builds a matrix from an input string, in case we want to specify an initial
    board layout.
def parseboard(boardstring):
    boardstring = string.replace(boardstring, ",", "")
    board, line = [], []
    for char in boardstring:
        if char == " ":
            line.append(None)
        elif char == "\n":
```

```python
            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

# Performs a move according to a given command, and returns the new game state.
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
        successor = copy.deepcopy(board)
        successor[y + dy][x + dx] = successor[y][x]
        successor[y][x] = None
        return successor
    else:
        raise ValueError("The move " + command + " by " + player.__class__.__name__ + " is not legal")

# Parse command-line arguments. See submitted report for summary of usage.
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", choices=["w", "b"], help="Play with a human opponent")
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")
parser.add_argument("-t", "--time", help="Timeout limit in seconds")
args = parser.parse_args()

cutoff = int(args.cutoff) if args.cutoff else 3
useab = not (args.alg == "mm") # Alpha-beta by default
logthegame = args.log
countingstates = args.count
fancy = args.fancy
timeout = float(args.time) if args.time else float("inf")

# If we give an input file, parse it and set up the initial board layout accordingly.
if args.input:
    with open(args.input, "r") as inputfile:
        initstr = inputfile.read()
    board = parseboard(initstr)
else: # If not, default to the starting position from the assignment.
    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"]
    ]

# Player instances
white = White()
black = Black()

# Designate a human player to a color if one is given, else let the computer play against itself.
if args.human == "w":
    human = white
```

```
    computer = black
elif args.human == "b":
    human = black
    computer = white
else:
    human = None
    computer = black # Arbitrary choice

# Other administrivia
currentplayer = white
log = ["Initial state:"]
movenumber = 1

# Main loop. Runs as long as there is no winner, or until interrupted.
while winner(board) is None:
    # Print informative stuff at the beginning of each round.
    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 = "" # Command string, e.g. 11E or 54N

    try: # In case of keyboard interrupts
        # Show a list of options to human players:
        if currentplayer is human:
            print "Possible moves:"
            for s in successors(board, currentplayer):
                print s.command
            cmd = raw_input()
        # Otherwise, have the computer calculate its move using the given algorithm.
        else:
            t = time.time() # Time limit is 20 seconds
            succs = successors(board, currentplayer) # Successors of this state
```

```
            # Take the first move, pick something better later on if we can find it.
            bestmove = succs[0].command
            bestutility = 0

            # Pick algorithm according to --alg argument.
            if useab: # Alpha-beta pruning
                for succ in succs:
                    # Initialize with alpha = -inf, beta = inf
                    u = alphabeta(currentplayer, succ, 0, float("-inf"), float("inf"))
                    if u > bestutility:
                        bestutility = u
                        bestmove = succ.command
                    if time.time() - t > timeout:
                        print "Time limit cutoff"
                        break
            else: # Minmax
                for succ in succs:
                    u = minmax(currentplayer, succ, 0)
                    if u > bestutility:
                        bestutility = u
                        bestmove = succ.command
                    if time.time() - t > timeout:
                        print "Time limit cutoff"
                        break

            cmd = bestmove
            print "The computer makes the move", cmd
            print "Thinking took", time.time() - t, "seconds"
            if logthegame:
                log.append("Thinking took " + str(time.time() - t) + " seconds")

        # May raise a ValueError if input is ill-formed.
        board = move(cmd, board, currentplayer)

        if countingstates:
            print statesvisited
            raise Exception("Counting states only, stopping here")
        if logthegame:
```

```
            log.append("%s plays %s" % (playername, cmd))

        # Move to next round
        currentplayer = white if currentplayer is black else black
        playername = currentplayer.__class__.__name__
        movenumber += 1

    # Catch errors made by user when entering a command:
    except ValueError:
        print "Illegal move."
    # Possibility for interrupting computation it takes too long.
    except KeyboardInterrupt:
        if logthegame:
            log.append("Game cancelled.")
        break

# Post-game formalities: print the board one last time, logging
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))
```