Assignment 1

Håkon Mork

ESCE 526 Artificial Intelligence

February 2, 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	694,652	
Improvement	$\times$18.76	$\times$26.49		

Game B:

Cutoff depth	3	4	5	6
Minmax	98,345	1,704,319		
α-β pruning	6,421			
Improvement	$\times$15.32			

Game C:

Cutoff depth	3	4	5	6
Minmax	69,954			
α-β pruning	3,763	51,098		
Improvement	$\times$18.59			

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

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. This 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 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`: Indicate that 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`.

Example:

```
python ass1.py --input file.txt --human --alg ab --cutoff 4
```

A.3 Listing

```
1 #!/usr/bin/env python
2
3 import string, copy, time, logging, argparse
4
5 # debug < info < warning < error < critical?
6 logging.basicConfig(level=logging.CRITICAL)
7
8 #withhuman = False # human vs. computer, or computer against itself
```

```
logthegame = False # write a log file on exit
fancy = False # simple or fancy heuristic

statesvisited = 0

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

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)
```

```
48                          succs.append(Node(candidate, player, cmd))
49                          #print "works ->", len(succs)
50                      except (ValueError, IndexError) as e:
51                          # ValueError: attempted move was illegal, e.g. trying
                                to move to an occupied square
52                          # IndexError: try to move outside of the board
53                          #print "".join(e)
54                          continue
55      logging.debug("There were " + str(len(succs)) + " successors")
56      return succs
57
58  def alphabeta(player, node, depth, alpha, beta):
59      global statesvisited
60      statesvisited += 1
61      succs = successors(node.board, player)
62      otherplayer = black if player is white else black
63      logging.info("Inside alphabeta on node " + str(hash(node)) + " obtained
            by " + node.command)
64      logging.info(str(hash(node)) + " looks like\n" + prettyprint(node.board))
65      logging.info(str(hash(node)) + " has depth = " + str(depth) + ", children
            = " + str(len(succs)))
66      logging.debug("They are (" + player.__class__.__name__ + "): ")
67      logging.debug("\n".join([c.command + " -> node " + str(hash(c)) for c in
            succs]))
68      if depth == cutoff or len(succs) == 0:
69          logging.info("Bottom reached, return utility " + str(node.value) + "
                from " + str(hash(node)))
70          if node.value > 0:
71              logging.info("Win found:\n" + prettyprint(node.board))
72          return node.value
73      elif player is white: #maxplayer, arbitrary
74          logging.debug("State is \n" + prettyprint(node.board))
75          for childnode in succs:
76              logging.debug("Entering examination of child " +
                    str(hash(childnode)) + " by " + childnode.command + " from "
                    + str(hash(node)))
77              alpha = max(alpha, alphabeta(otherplayer, childnode, depth + 1,
                    alpha, beta))
78              if alpha >= beta:
79                  logging.info("Pruning: returning beta = " + str(beta) + "
                        from " + str(hash(childnode)))
```

```
80                  return beta
81          logging.info("No pruning: returning alpha = " + str(alpha) + " from "
                + str(hash(node)))
82          return alpha
83      else: #black minplayer
84          logging.debug("State is \n" + prettyprint(node.board))
85          for childnode in succs:
86              logging.debug("Entering examination of child " +
                    str(hash(childnode)) + " by " + childnode.command + " from "
                    + str(hash(node)))
87              beta = min(beta, alphabeta(otherplayer, childnode, depth + 1,
                    alpha, beta))
88              if alpha >= beta:
89                  logging.info("Pruning: returning alpha = " + str(alpha) + "
                        from " + str(hash(childnode)))
90                  return alpha
91          logging.info("No pruning: returning beta = " + str(beta) + " from " +
                str(hash(node)))
92          return beta
93
94  def minmax(player, node, depth):
95      global statesvisited
96      statesvisited += 1
97      logging.debug("Inside minmax on node " + str(hash(node)) + " depth = " +
            str(depth))
98      #otherplayer = white if player is black else black
99      minplayer = black # arbitrary
100     if depth == cutoff or not successors(node.board, player):
101         logging.debug("Bottom reached, return utility " + str(node.value))
102         if node.value > 0:
103             logging.debug("Win found:\n" + prettyprint(node.board))
104         return node.value
105     elif node.player is minplayer:
106         logging.debug("Recursive minmax: player " + str(player) + ", depth =
                " + str(depth) + ", node = " + str(hash(node)))
107         return min(minmax(player, child, depth + 1) for child in
                successors(node.board, player))
108     else:
109         logging.debug("Recursive minmax: player " + str(player) + ", depth =
                " + str(depth) + ", node = " + str(hash(node)))
```

```
110         return max(minmax(player, child, depth + 1) for child in
                successors(node.board, player))
111
112 def prettyprint(board):
113     b = "\n".join(",".join(map(str, row)) for row in board)
114     return b.replace("None", " ")
115
116 def winner(board):
117     # indicate the winner (if any) in the given board state
118     def horizontal(board):
119         # check if any consecutive four entries in a row are X-es or O-s
120         for line in board:
121             for i, char in enumerate(line):
122                 if line[i : i + 4] == ["O"] * 4:
123                     return white
124                 elif line[i : i + 4] == ["X"] * 4:
125                     return black
126     def vertical(board):
127         # equivalent to the horizontal winner in the transposed matrix
128         return horizontal(map(list, zip(*board)))
129     def diagonal(board):
130         # all downward diagonals must start in the upper-left 4x4 submatrix
131         # similarly, all upward diagonals must start in the lower-left 4x4
                submatrix
132         # somewhat inelegant, but it works
133         for i in range(4):
134             for j in range(4):
135                 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)):
136                     return white
137                 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)):
138                     return black
139     return horizontal(board) or vertical(board) or diagonal(board)
140
141 def simpleheuristic(board, player):
142     otherplayer = white if player is black else black
143     if winner(board) is player:
144         return 1
145     elif winner(board) is otherplayer:
146         return -1
```

```
147     else:
148         return 0
149 
150 def fancyheuristic(board, player):
151     pass
152 
153 def parse(boardstring):
154     # build a matrix from a string describing the board layout
155     boardstring = string.replace(boardstring, ",", "")
156     board, line = [], []
157     for char in boardstring:
158         if char == " ":
159             line.append(None)
160         elif char == "\n":
161             board.append(line)
162             line = []
163         else:
164             line.append(char)
165     if line:
166         board.append(line) # last line, if there is no newline at the end
167     return board
168 
169 
170 def move(command, board, player):
171     # takes indices and a direction, e.g. "43W" or "26N"
172     x, y, d = tuple(command)
173     # the board is a zero-indexed array, adjust accordingly
174     x, y = int(x) - 1, int(y) - 1
175     dy, dx = directions[d.upper()]
176     # does the piece fall within the bounds?
177     if ((0 <= x + dx <= 7) and (0 <= y + dy <= 7)
178     # and is it our piece?
179     and board[y][x] == player.piece
180     # and is the destination square empty?
181     and not board[y + dy][x + dx]):
182         # then it's okay
183         # we don't want to update in place
184         successor = copy.deepcopy(board)
185         successor[y + dy][x + dx] = successor[y][x]
186         successor[y][x] = None
187         return successor
```

```
188     else:
189         raise ValueError#("The move " + command + " is not legal")
190 
191 white = White()
192 black = Black()
193 computer = black
194 currentplayer = white
195 #cutoff = 4
196 
197 parser = argparse.ArgumentParser()
198 parser.add_argument("-c", "--cutoff", help="Cutoff depth")
199 parser.add_argument("-i", "--input", help="Input game board")
200 parser.add_argument("-u", "--human", help="Play with a human opponent")
201 parser.add_argument("-a", "--alg", choices=["mm", "ab"], help="Minmax or
        alpha-beta algorithm")
202 args = parser.parse_args()
203 
204 cutoff = int(args.cutoff) if args.cutoff else 3
205 human = white if args.human else None
206 useab = (args.alg == "ab")
207 
208 if args.input:
209     with open(args.input, "r") as inputfile:
210         initstr = inputfile.read()
211     board = parse(initstr)
212 else:
213     board = [
214         ["O", None, None, None, None, None, "X"],
215         ["X", None, None, None, None, None, "O"],
216         ["O", None, None, None, None, None, "X"],
217         ["X", None, None, None, None, None, "O"],
218         ["O", None, None, None, None, None, "X"],
219         ["X", None, None, None, None, None, "O"],
220         ["O", None, None, None, None, None, "X"]
221     ]
222 
223 # with open("./startb.txt", "r") as f:
224 #   initstatestr = f.read()
225 # board = parse(initstatestr)
226 
227 #board = initialstate
```

```
228 log = ["Initial state:"]
229 movenumber = 1
230
231 while winner(board) is None:
232     playername = currentplayer.__class__.__name__
233     p = prettyprint(board)
234     print p
235     log.append(p)
236     print "\nMove #%s:" % movenumber
237     log.append("\nMove #%s:" % movenumber)
238     cmd = ""
239     print "It's %s's turn." % playername
240     try:
241         if currentplayer is human:
242             print "Possible moves:"
243             for s in successors(board, currentplayer):
244                 print s.command
245             cmd = raw_input()
246         else: #let the computer play against itself
247             succs = successors(board, currentplayer)
248             # take the possible move now, pick something better later on if
                  we can find it
249             bestmove = succs[0].command
250             bestutility = 0
251             if useab: #alphabeta
252                 logging.warning("Player " + playername + " thinking about
                      what to do.")
253                 logging.warning("Using alphabeta with cutoff " + str(cutoff))
254                 for succboard in succs:
255                     #init with alpha = -inf, beta = inf
256                     u = alphabeta(currentplayer, succboard, 0, float("-inf"),
                          float("inf"))
257                     if u > bestutility:
258                         bestutility = u
259                         bestmove = succboard.command
260             else: #minmax
261                 logging.warning("Player " + playername + " thinking about
                      what to do.")
262                 logging.warning("Using minmax with cutoff " + str(cutoff))
263                 for succboard in succs:
264                     u = minmax(currentplayer, succboard, 0)
```

```
265                     if u > bestutility:
266                         logging.critical("Utility improved: " + str(u) + "
                               from " + succboard.command)
267                         bestutility = u
268                         bestmove = succboard.command
269             cmd = bestmove
270             print "The computer makes the move", cmd
271 
272         print "cutoff", cutoff, "states", statesvisited, "with", "alphabeta"
               if useab else "minmax"
273         raise Exception("Counting states visited")
274         board = move(cmd, board, currentplayer)
275         log.append("%s plays %s." % (playername, cmd))
276         currentplayer = white if currentplayer is black else black
277         playername = currentplayer.__class__.__name__
278         movenumber += 1
279     #except ValueError:
280     #   print "Illegal move."
281         #raise
282     except KeyboardInterrupt:
283         log.append("Game cancelled.")
284         logging.critical("Game cancelled.")
285         break
286 
287 # post-game cleanup
288 print prettyprint(board)
289 log.append(prettyprint(board))
290 
291 if winner(board):
292     s = "%s won the match" % winner(board).__class__.__name__
293     print s
294     log.append(s)
295 else:
296     print "It's a draw"
297     log.append("It's a draw")
298 
299 if logthegame:
300     logname = time.strftime("/Users/hakon/Desktop/con4-%Hh%M-%S.log")
301     with open(logname, "w+") as logfile:
302         logfile.write("\n".join(log))
```