Assignment 1

Håkon Mork

ESCE 526 Artificial Intelligence

January 31, 2013

1

1.1 Number of states visited

Cutoff	3	4	5	6
Minmax				
Alpha-beta pruning				

1.2 Does state generation order matter?

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

2.4 Gameplay log

Implementation comments

Appendix: Source code

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

import string, copy, time, logging

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

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

# 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 = simpleutility(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))
40      succs = dict()
41      for y, line in enumerate(board):
42          for x, char in enumerate(line):
43              if char == player.piece:
44                  # try all possible moves: xyN, xyE, xyS, xyW
45                  for cmd in (str(x + 1) + str(y + 1) + d for d in directions):
46                      #print player.__class__.__name__, cmd,
47                      try:
48                          candidate = move(cmd, board, player)
49                          succs[cmd] = Node(candidate, player, cmd)
50                          #print "works ->", len(succs)
51                      except (ValueError, IndexError) as e:
52                          # valueerror: attempted move was illegal, e.g. trying
                               to move to an occupied square
53                          # indexerror: try to move outside of the board
54                          #print "".join(e)
55                          continue
56      logging.debug("There were " + str(len(succs)) + " successors")
57      return succs
58
59  def alphabeta(player, node, depth, alpha, beta):
60      logging.info("Inside alphabeta on node " + str(hash(node)) + " obtained
            by " + node.command)
61      succs = successors(node.board, player)
62      logging.info(str(hash(node)) + " looks like\n" + prettyprint(node.board))
63      logging.info(str(hash(node)) + " has depth = " + str(depth) + ", children
            = " + str(len(succs)))
64      logging.info("They are (" + player.__class__.__name__ + "): " + "
            ".join([c for c in succs]))
65  #   logging.debug("Depth = " + str(depth) + ", children = " +
        str(len(node.succs)))
66      otherplayer = black if player is white else black
67      if depth == cutoff or len(succs) == 0:
68          logging.info("Bottom reached, return utility " + str(node.value) + "
                from " + str(hash(node)))
69          if node.value > 0:
70              logging.info("Win found:\n" + prettyprint(node.board))
71          return node.value
72      else:
73          logging.info("Recursive alphabeta by %s" % player.__class__.__name__)
```

```
74  #       logging.debug("State is \n" + prettyprint(node.board))
75          for childcmd, childnode in succs.items():
76              logging.info("Entering examination of child " +
                    str(hash(childnode)) + " by " + childcmd + " from " +
                    str(hash(node)))
77              #logging.info(str(hash(childnode)) + " looks like\n" +
                    prettyprint(childnode.board))
78              if player is white: #maxplayer
79                  alpha = max(alpha, alphabeta(otherplayer, childnode, depth +
                        1, alpha, beta))
80                  if alpha >= beta:
81                      logging.info("Pruning: returning beta = " + str(beta) + "
                            from " + str(hash(childnode)))
82                      return beta
83                      #break
84                  logging.info("No pruning: returning alpha = " + str(alpha) +
                        " from " + str(hash(childnode)))
85                  return alpha
86              else: #minplayer
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                      #break
92                  logging.info("No pruning: returning beta = " + str(beta) + "
                        from " + str(hash(childnode)))
93                  return beta
94      #elif player is black: #maxplayer (arbitrary)
95
96      # else: #minplayer (arbitrary)
97      #   logging.debug("Recursive alphabeta by minplayer Black")
98      #   logging.debug("State is \n" + prettyprint(node.board))
99      #   for childcmd, childnode in successors(node.board, player).items():
100     #
101     #
102
103 def minmax(player, node, depth):
104     logging.debug("Inside minmax on node " + str(hash(node)))
105     #otherplayer = white if player is black else black
```

```
106     minplayer = black # arbitrary
107     if depth == cutoff or not successors(node.board, player):
108         logging.debug("Bottom reached, return utility " + str(node.value))
109         if node.value > 0:
110             logging.debug("Win found:\n" + prettyprint(node.board))
111         return node.value
112     elif node.player is minplayer:
113         logging.debug("Recursive minmax: player " + str(player) + ", depth =
                " + str(depth) + ", node = " + str(hash(node)))
114         return min(minmax(player, child, depth + 1) for child in
                successors(node.board, player).values())
115     else:
116         logging.debug("Recursive minmax: player " + str(player) + ", depth =
                " + str(depth) + ", node = " + str(hash(node)))
117         return max(minmax(player, child, depth + 1) for child in
                successors(node.board, player).values())
118
119 def prettyprint(board):
120     b = "\n".join(",".join(map(str, row)) for row in board)
121     return b.replace("None", " ")
122
123 def winner(board):
124     # indicate the winner (if any) in the given board state
125     def horizontal(board):
126         # check if any consecutive four entries in a row are X-es or O-s
127         for line in board:
128             for i, char in enumerate(line):
129                 if line[i : i + 4] == ["O"] * 4:
130                     return white
131                 elif line[i : i + 4] == ["X"] * 4:
132                     return black
133     def vertical(board):
134         # equivalent to the horizontal winner in the transposed matrix
135         return horizontal(map(list, zip(*board)))
136     def diagonal(board):
137         # all downward diagonals must start in the upper-left 4x4 submatrix
138         # similarly, all upward diagonals must start in the lower-left 4x4
                submatrix
139         # somewhat inelegant, but it works
140         for i in range(4):
141             for j in range(4):
```

```
142                 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)):
143                     return white
144                 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)):
145                     return black
146     return horizontal(board) or vertical(board) or diagonal(board)
147
148 def simpleutility(board, player):
149     otherplayer = white if player is black else black
150     if winner(board) is player:
151         return 1
152     elif winner(board) is otherplayer:
153         return -1
154     else:
155         return 0
156
157 def fancyutility(board, player):
158     pass
159
160 def parse(boardstring):
161     # build a matrix from a string describing the board layout
162     boardstring = string.replace(boardstring, ",", "")
163     board, line = [], []
164     for char in boardstring:
165         if char == " ":
166             line.append(None)
167         elif char == "\n":
168             board.append(line)
169             line = []
170         else:
171             line.append(char)
172     if line:
173         board.append(line) # last line, if there is no newline at the end
174     return board
175
176
177 def move(command, board, player):
178     # takes indices and a direction, e.g. "43W" or "26N"
179     x, y, d = tuple(command)
180     # the board is a zero-indexed array, adjust accordingly
```

```
181      x, y = int(x) - 1, int(y) - 1
182      dy, dx = directions[d.upper()]
183      # does the piece fall within the bounds?
184      if ((0 <= x + dx <= 7) and (0 <= y + dy <= 7)
185      # and is it our piece?
186      and board[y][x] == player.piece
187      # and is the destination square empty?
188      and not board[y + dy][x + dx]):
189          # then it's okay
190          # we don't want to update in place
191          successor = copy.deepcopy(board)
192          successor[y + dy][x + dx] = successor[y][x]
193          successor[y][x] = None
194          return successor
195      else:
196          raise ValueError#("The move " + command + " is not legal")
197
198  white = White()
199  black = Black()
200  human = white if withhuman else None
201  computer = black
202  currentplayer = white
203  cutoff = 4
204
205  # tuples of (dy, dx) for all directions
206  directions = {
207      "N": (-1, 0),
208      "E": (0, 1),
209      "S": (1, 0),
210      "W": (0, -1)
211  }
212
213  with open("./starta.txt", "r") as f:
214      initstatestr = f.read()
215  board = parse(initstatestr)
216
217  #board = initialstate
218  log = ["Initial state:"]
219  movenumber = 1
220
221  while winner(board) is None:
```

```
222     playername = currentplayer.__class__.__name__
223     p = prettyprint(board)
224     print p
225     log.append(p)
226     print "\nMove #%s:" % movenumber
227     log.append("\nMove #%s:" % movenumber)
228     cmd = ""
229     print "It's %s's turn." % playername
230     try:
231         if currentplayer is human:
232             print "Possible moves:"
233             for s in successors(board, currentplayer):
234                 print s.command
235             cmd = raw_input()
236         else: #let computer play against itself
237             succs = successors(board, currentplayer)
238             if useab: #alphabeta
239                 logging.info("Player " + playername + ", using alphabeta")
240                 bestutility = 0
241                 bestmove = None
242                 for succmove, succboard in succs.items():
243                     #init with alpha = -inf, beta = inf
244                     u = alphabeta(currentplayer, succboard, 0, float("-inf"),
                            float("inf"))
245                     if u > bestutility:
246                         bestutility = u
247                         bestmove = succmove
248                 #if not bestmove:
249                 #   logging.warning("Move command is None")
250                 logging.info("Best move: " + bestmove + " with utility " +
                        str(bestutility))
251             else: #minmax
252                 logging.info("Using minimax")
253                 bestutility = 0
254                 bestmove = None
255                 for succmove, succboard in succs.items():
256                     u = minmax(currentplayer, succboard, 0)
257                     if u > bestutility:
258                         bestutility = u
259                         bestmove = succmove
260                 #utilities = [minmax(currentplayer, succ, cutoff) for succ in
```

```
                    succs.values()]
261                 #print utilities
262
263             # for s in successors(board, currentplayer).values():
264             #   print s.command, s.value
265
266             cmd = bestmove
267             print "The computer makes the move", cmd
268
269         board = move(cmd, board, currentplayer)
270         log.append("%s plays %s." % (playername, cmd))
271         currentplayer = white if currentplayer is black else black
272         playername = currentplayer.__class__.__name__
273         movenumber += 1
274     #except ValueError:
275     #   print "Illegal move."
276         #raise
277     except KeyboardInterrupt:
278         log.append("Game cancelled.")
279         logging.warning("Game cancelled.")
280         break
281
282 # post-game cleanup
283 print prettyprint(board)
284 log.append(prettyprint(board))
285
286 if winner(board):
287     s = "%s won the match" % winner(board).__class__.__name__
288     print s
289     log.append(s)
290 else:
291     print "It's a draw"
292     log.append("It's a draw")
293
294 if logthegame:
295     logname = time.strftime("/Users/hakon/Desktop/connect4-%Hh%M-%S.log")
296     with open(logname, "w+") as logfile:
297         logfile.write("\n".join(log))
```