Tic-Tac-Toe with Minimax

AI – Gabor – 14/15 Dec 2021

```
def minimax(brd, tkn):
  # returns a dictionary of mv:WinDrawLoss (1,0, or -1) for x
  res = {}                                  # Result dictionary
  eTkn = …                                  # x->o, o->x
  for mv in moves(brd):
    newBrd = makeMove(brd,tkn,mv)
    eRes = minimax(newBrd,eTkn)             # Next level's results
    if tkn == "x":
      if any([eRes[m]==-1 for m in eRes]):   res[mv] = -1  # min win
      elif any([not eRes[m] for m in eRes]): res[mv] = 0   # tie
      else:                                  res[mv] = 1   # min loses
    else:
      if any([eRes[m]==1 for m in eRes]):    res[mv] = 1   # max win
      elif any([not eRes[m] for m in eRes]): res[mv] = 0   # tie
      else:                                  res[mv] = -1  # max loses
  return res
```

What is wrong with the above? How can we fix it?

```
def minimax(brd, tkn):
  # returns a dictionary of mv:WinDrawLoss (1,0, or -1) for x
  # Terminating conditions
  triples = {brd[0:3], brd[3:6], brd[6:9], brd[0:9:3],
             brd[1:9:3], brd[2:9:3], brd[0:9:4], brd[2:7:2]}
  if "ooo" in triples:   return {"": -1}
  if "xxx" in triples:   return {"": 1}
  if not brd.count("."): return {"": 0}

  res = {}                                  # Result dictionary
  eTkn = …                                  # x->o, o->x
  for mv in moves(brd):
    newBrd = makeMove(brd,tkn,mv)
    eRes = minimax(newBrd,eTkn)             # Next level's results
    if tkn == "x":
      if any([eRes[m]==-1 for m in eRes]):   res[mv] = -1  # min win
      elif any([not eRes[m] for m in eRes]): res[mv] = 0   # tie
      else:                                  res[mv] = 1   # min loses
    else:
      if any([eRes[m]==1 for m in eRes]):    res[mv] = 1   # max win
      elif any([not eRes[m] for m in eRes]): res[mv] = 0   # tie
      else:                                  res[mv] = -1  # max loses
  return res
```

Let's tidy up the recursive part:

```
def minimax(brd, tkn):
  # returns a dictionary of mv:WinDrawLoss (1,0, or -1) for x
  # Terminating conditions
  triples = {brd[0:3], brd[3:6], brd[6:9], brd[0:9:3],
             brd[1:9:3], brd[2:9:3], brd[0:9:4], brd[2:7:2]}
  if "ooo" in triples: return {"": -1}
  if "xxx" in triples: return {"": 1}
  if "." not brd:      return {"": 0}

  res = {}                                # Result dictionary
  eTkn = …                                # x->o, o->x
  for mv in moves(brd):
    newBrd = makeMove(brd,tkn,mv)
    eRes = minimax(newBrd,eTkn)           # Next level's results
    if tkn == "x":
      res[mv] = min(eRes.values())        # The best that o can do
    else:
      res[mv] = max(eRes.values())        # The best that x can do
  return res
```

We can move the loop into a comprehension:

```
def minimax(brd, tkn):
  # returns a dictionary of mv:WinDrawLoss (1,0, or -1) for x
  # Terminating conditions
  triples = {brd[0:3], brd[3:6], brd[6:9], brd[0:9:3],
             brd[1:9:3], brd[2:9:3], brd[0:9:4], brd[2:7:2]}
  if "ooo" in triples: return {"": -1}
  if "xxx" in triples: return {"": 1}
  if "." not brd:      return {"": 0}

  res = {}                                # Result dictionary
  eTkn = …                                # x->o, o->x
  if tkn == "x":
    return {mv:min(minimax(makeMove(brd,tkn,mv),eTkn).values())
            for mv in moves(brd)}
  else:
    return {mv:max(minimax(makeMove(brd,tkn,mv),eTkn).values())
            for mv in moves(brd)}
```

Finally, we can consolidate the triples, and also the two return statements:

```
# base, diff
BSDF = [(0,1), (3,1), (6,1), (0,3), (1,3), (2,3), (0,4), (2,2)]

def minimax(brd, tkn):
  # returns a dictionary of mv:WinDrawLoss (1,0, or -1) for x
  # Terminating conditions
  triples = {brd[bs:bs+3*df:df] for bs, df in BSDF}
  if "ooo" in triples: return {"": -1}
  if "xxx" in triples: return {"": 1}
  if "." not brd:      return {"": 0}

  res = {}                                # Result dictionary
  eTkn = …                                # x->o, o->x
  nx = min if tkn == "x" else max
  return {mv:nx(minimax(makeMove(brd,tkn,mv),eTkn).values())
             for mv in moves(brd)}
```

Introducing Negamax, minimax's cooler cousin

Negamax applies when the game evaluation is symmetric with respect to 0. In other words, where the valuations would be opposite of each other if you swapped the tokens. Minimax evaluates with respect to one specific token (White in chess, 'x' in Tic-Tac-Toe, typically the token that moves first).

Negamax evaluates with respect to the token whose move it is. Each token is trying to get a maximally positive score. It looks like this:

```
def negamax(brd, tkn):
  # returns dict with keys one of "W","L","D" (win, loss, draw)
  # vals are the set of moves leading to that condition

  triples = {brd[bs:bs+3*df:df] for bs, df in BSDF}
  eTkn = …
  if game is over: return …

  res = {"W":set(), "D":set(), "L":set()}
  for mv in moves(brd):
    nm = negamax(makeMove(brd,tkn,mv), eTkn)
    mvCategory = …
    res[mvCategory].add(mv)

  return res
```

For the terminating consequences and mvCategory, what about?

```
def negamax(brd, tkn):
  # returns dict with keys one of "W","L","D" (win, loss, draw)
  # vals are the set of moves leading to that condition

  triples = {brd[bs:bs+3*df:df] for bs, df in BSDF}
  eTkn = …
  if 3*tkn in triples: return {"W":…, "D":set(), "L":set()}
  if "." not in brd:   return {"W":set(), "D":…, "L":set()}

  res = {"W":set(), "D":set(), "L":set()}
  for mv in moves(brd):
    nm = negamax(makeMove(brd,tkn,mv), eTkn)
    mvCategory = "L" if nm["W"] else ("D" if nm["D"] else "W")
    res[mvCategory].add(mv)

  return res
```

Now we can fill in the two ellipses:

```
def negamax(brd, tkn):
  # returns dict with keys one of "W","L","D" (win, loss, draw)
  # vals are the set of moves leading to that condition

  triples = {brd[bs:bs+3*df:df] for bs, df in BSDF}
  eTkn = …
  if 3*tkn in triples: return {"W":"Hi",  "D":set(), "L":set()}
  if "." not in brd:   return {"W":set(), "D":"mom", "L":set()}

  res = {"W":set(), "D":set(), "L":set()}
  for mv in moves(brd):
    nm = negamax(makeMove(brd,tkn,mv), eTkn)
    mvCategory = "L" if nm["W"] else ("D" if nm["D"] else "W")
    res[mvCategory].add(mv)

  return res
```

Too bad, the above does not work. Can you see why?

```
def negamax(brd, tkn):
  # returns dict with keys one of "W","L","D" (win, loss, draw)
  # vals are the set of moves leading to that condition

  triples = {brd[bs:bs+3*df:df] for bs, df in BSDF}
  eTkn = …
  if 3*eTkn in triples: return {"W":set(), "D":set(), "L":"Hi"}
  if "." not in brd:    return {"W":set(), "D":"mom", "L":set()}

  res = {"W":set(), "D":set(), "L":set()}
  for mv in moves(brd):
    nm = negamax(makeMove(brd,tkn,mv), eTkn)
    mvCategory = (nm["W"] and "L") or (nm["D"] and "D") or "W"
#   mvCategory = "L" if nm["W"] else ("D" if nm["D"] else "W")
    res[mvCategory].add(mv)

  return res
```

Othello 5 – The Negamax lab

AI – Gabor – 14/15 Dec 2021

We'd like to apply minimax or Negamax to Othello. Because the game tree for Othello is much greater than 3x3 tic-tac-toe, we will first focus on applying Negamax to the endgame. This way, there is no need for board estimation (ie. we will use the real truth – the difference in the number of tokens captured) while we get the details of Negamax down.

In theory, we could take the tic-tac-toe code almost directly because proper Othello only cares about wins, draws, and losses. In this class, however, we care about the number of tokens captured. Thus, it doesn't make sense to have only 3 keys, so for each move we'll determine the best score that can be achieved.

You should keep what you have done for Othello 4. However, in the case where the number of open positions is less than N (where N=11 for now), you will run an additional piece of code. This code will run Negamax (or minimax) and determine a guaranteed minimum score, and an optimal move sequence which will guarantee it (optimal means both sides play optimally). You should output the word **score**, then this guaranteed minimum score (possibly negative), followed by a sequence of moves IN REVERSE which will lead to the score. The score is from the point of view of the token given (rather than the number of x tokens minus the number of o tokens). The reason for the reverse sequence is that this way, the final move is what will be picked up by the moderator as the script's best move. The sequence of moves includes both your script's moves and the moves of the opponent. Negative numbers will be ignored, so passes may (and should) be indicated by a -1.

Scoring is .5 for a (there may be more than one) correct Next move (ie. the final integer output), .25 for a correct Minimum score, and .25 for a correct Move sequence. Note that just because a move sequence leads to the correct score does not mean that the move sequence is correct (see below for details).

The grader will first test the submitted script on 10 fixed boards, where the number of free positions increases from 1 to 10. Then, the grader will take 9 or 10 random board positions with one free spot, then 9 or 10 with 2 free spots, and so on until 100 tests have been run. Currently, the 10th test is the only one where there are 10 free spots left on the Othello board. The grader will deliver a board and a token to your script, where there is at least one move by that token. The grader will take the most recent line of the script's output which has the word "score" and at least one number (if it does not find such a line, it will assume the script has timed out - see below for what the grader will then do). The first number on this line is the minimum score that the given token is guaranteed to achieve under perfect play. The remaining numbers, in reverse order, constitute a sequence of moves that will achieve this score. If there is a script or syntax error, or the final number the grader sees is not a valid move, then the grading script will summarily terminate.

If the script being tested does put the word score on one line, along with at least one number, then the grader script will find the last number in the output and take that as its move. In prior testing,

the likelihood of optimal rule-of-thumb moves near the end of the game is around 50% (ie. the midgame rules of thumb are not so good in the endgame). On the plus side, the testing is faster :)

IMPORTANT PRACTICAL NOTES: Initially, this lab is about accuracy, then it becomes about speed. To that end, at first you should be concerned with getting Negamax (or minimax) working when there are only a few open places left on the board (as in less than 5). Once you have that fixed up, then you can worry about making your code fast enough. It's essentially like tic-tac-toe, only trickier, because it might be that one side or another has to pass (a common situation). It might also be, though not as common, that there are empty spaces left over at the end.

The grader looks to see if it can find the word "score" on any line, and if so, it extracts integers on that line as the score followed by the reversed sequence which would achieve that score (such sequence is not necessarily unique). If it does not find the word score, or there are no numbers on that line, then the grader will find the last integer in the entire output and assume that is the move selected by the submitted script (with no score and no move sequence given).

Note: If you output timing information, that is OK since it will be a non-integer (number of seconds). However, printing out a date is not so good because that will have at least one integer in it.

Even if you do not do Negamax (ie. you just do a rule-of-thumb), the tests near the end will take longer because the grader script has to do its own Negamax in order to evaluate the output of the submitted script. If the grader runs through to the end, it will show the two most offending tests (if any). Since this lab is about Negamax, it will not show rule-of-thumb outputs (to view those logs, ask me in person). In other words, to get feedback from the grader, you must have the Negamax output format in place (the word score, followed by a minimum score, followed by a reversed move sequence on one line).

There are eight possible symbols that might appear as the script is being run (rule-of-thumb: + and capital letters are better than - and lowercase letters):

`+:` successful test (+1)
`M:` Score and Next move are correct, but Move sequence is wrong (+3/4)
`Q:` Move sequence is correct, but Score is wrong (+3/4)
`T:` Timeout (or no Negamax), with correct Next move (+1/2)
`m:` Next move is correct, but Score and Move sequence are wrong (+1/2)
`s:` Score is correct, but Next move is wrong (+1/4)
`-:` Neither the score nor the Next move is correct (0)
`t:` Timeout (or no Negamax), with incorrect Next move (0)

Example test run -- note that the corner move for x would actually be losing -- tn stands for terminal Negamax. Also, everything through the "Possible moves ..." line is Othello 3, through the "My corner ..." line is Othello 4, and the "Min score ..." line is Othello 5:

```
>tn-pcgabor.py xxxxxxo.xxxxxo..xxooooooxoxxooooxoxxoooxxxoxoooxxo.oxooxooooooo x
xxxxxxo*
xxxxxo*.
xxooooo
xoxxoooo
xoxxoooo
xxxoxooo
xxo*oxoo
xoooooo

xxxxxxo.xxxxxo..xxooooooxoxxooooxoxxoooxxxoxoooxxo.oxooxooooooo  27/33
Possible moves for x: 7, 14, 51
My corner move is 7
Min score: 18; move sequence: [15, -1, 7, -1, 14, -1, 51]
Elapsed time: 0.010512828826904297
```

An example of the s phenomenon -- the move sequence in the following example is incorrect:

```
>tn-pcgabor.py xxxxxxo.xxxoxo.oxxxxoooooxoxxoox.xxoxxxxxxxxxxoxxoxxxxxxx.xxxxxxx. o

xxxxxxo.
xxxoxo.o
xxxxoooo
xoxxoox*
xxoxxxxx
xxxxoxxo
xxxxxxx*
xxxxxxx.

xxxxxxo.xxxoxo.oxxxxoooooxoxxoox.xxoxxxxxxxxxxoxxoxxxxxxx.xxxxxxx.  45/14

Possible moves for o: 31, 55
My move is 31
Min score: -48; move sequence: [14 -1 63 55 7 31]
```

The score indicated for o is correct, and the reversed sequence of moves would lead to that score. However, if o were to play 31, then x would not play the foolish move 7. Instead, it would play 14, which would lead to a score for o of -51 and not -48. In other words, the move sequence that was given was suboptimal. It should, instead, be optimal from the point of view of both sides.