Othello Lab 6: From Negamax to Alpha-beta
AI - Jan 2021

Most game playing software uses some form of Minimax, such as Negamax, to analyze the game state, in order to decide what the best move is going forward.

We have seen pseudo code for Negamax, for Othello purposes, along the lines of:

```
def negamax(brd, tkn):
  if no possible moves for tkn:        # if tkn can't move
    if no possible moves for enemy:    #   if game is over
      return [the score]
    nm = recurse on negamax            #   game not over; tkn passes
    return the right thing

  best = min(negamax(makeMove(brk, tkn, mv), enemy) + [mv]
             for mv in possible moves for tkn)
  return [-best[0]] + best[1:]
```

The difference between Minimax and Negamax is that Minimax always evaluates the board from a fixed point of view (such as positive being good for white in chess, and for 'x' (or black) in Othello), while Negamax attempts to maximize things for the player in question (`tkn` in the case above)

We discussed some of these Negamax speedups and gotchas:

1) If you've been setting globals, even indirectly, `negamax()` will not play nice with your code. Recursion together with setting globals is a big no-no. Using globals as lookup tables is fine.
2) Cache values for `findMoves()`. Using globals for caching is fine, too.
3) Short circuit and return right away when the game is not over, there is exactly one move, and you can easily compute the score if the one move were to be made.
4) Split `negamax()` into two functions: a top level (public) function, and an internal one. The top level `negamax()` unwinds the comprehension so that it checks the result of each internal `negamaxR()` call (the R standing for recursive), and if it's an improvement on the prior one, then it prints out the improvement. You could, instead, avoid the split by using an implied (optional) parameter.
5) Probably `makeMove()` should not do individual token replacements on the string. If `brd` is a string (the case for most), blast the string to a list, do the token flips at the appropriate positions, and then reconstitute with a `"".join(…)`.
6) Cache values for `negamaxR()`.
7) If you are getting `T`s in your output this means that you guessed the right move but got timed out. This likely means you are leaving points on the table. In particular, it means that your Othello 4 code has been good to you, but you didn't return the favor by having that move be the starting out move for `negamax()`.

Alpha-beta is a way of implementing Minimax or Negamax to get even more speed out of it, which translates into being able to run it to a greater depth. It is an enhancement (ie. improvement) on Minimax/Negamax, rather than being distinct. The idea is to establish that the score going down a particular branch must be within certain bounds to be of use, and if the actual score falls outside the specified bounds, to conclude that further investigation of the branch is not needed. The pseudo code for Alpha-beta applied to Negamax could be written as:

```
def alphabeta(brd, tkn, lowerBnd, upperBnd):
  if no possible moves for tkn:        # if tkn can't move
    if no possible moves for enemy:    #   if game is over
      return [the score]
    ab = recurse with alphabeta        # game not over
    return the right thing

  best = [lowerBnd-1]                  # guarantees best will be set
  for mv in possible moves for tkn:
    ab = alphabeta(makeMove(brd,tkn,mv), enemy, -upperBnd, -lowerBnd)
    score = -ab[0]                           # Score from tkn's viewpt
    if score < lowerBnd: continue            # Not an improvement
    if score > upperBnd: return [score]      # Vile to the caller
    update best                              # Else it's an improvement
    lowerBnd = score+1

  return best
```

This code has some analogy to the top level version of Negamax in that the comprehension in the main body has been unrolled to a loop. In particular, in the loop, as one gets back a score from `alphabeta()` going down a particular branch, after converting it from the enemy's point of view to that of `tkn`, there are three scenarios. The first scenario, which also holds in Negamax sans Alpha-beta, is that the score is not better than the best score from a prior branch. In other words, `tkn` has done better earlier; therefore, this new branch is not of interest to `tkn`.

The remaining scenarios, that the returned score from the current branch improves upon the prior best score, can be divided into two distinct parts. The usual case here, which also holds with Negamax, is when the improvement is moderate, within the bounds of "reasonableness". "Reasonableness" here is defined by `upperBnd`. In this case, the code updates what it intends to return (`best`) and updates the `lowerBnd` (that is to say, this branch is now the best branch), and continues processing.

However, in the remaining option, which only holds with Alpha-beta, there can be a case of too-much-of-a-good-thing (meaning that the score is greater than `upperBnd`). This is where the branch is so good that `tkn` would be overjoyed to have it, but the enemy has already seen a branch (higher up) that is better for it (and less good for `tkn`), precluding this *entire set* of branches. Specifically, all remaining branches that have not yet been tried by `tkn` are ignored (pruned). Note that `[score]` is returned sans moves. This is since we already know the caller will repudiate this branch (based on the score) and never use it.

Possible speedups to Alpha-beta:

1) `findMoves()` can still be cached, but `alphabeta()` is not so amenable because of the two additional parameters.

2) The short circuit described for Negamax also works here.

3) Splitting `alphabeta()` into two functions, a top level one (taking only brd and tkn as parameters and doing a print with each improvement) and an internal one, shown above, improves one's chances at the point where Alpha-beta starts to suffer from time constraints.

4) The order in which moves is tried is relevant. The better the move ordering for a branch, the more likely it is that pruning will happen, since the upper and lower bounds will be closer. When near the end of the game, rule-of-thumb ordering is not so effective, but it can be when there are many holes. For example, one could order by putting corners first and then non-corners.

5) There are many ways to implement `findMoves(brd, tkn)`. One way is to go through the entire `brd`, examining each position, and do further examination if there is a hole at that position. Since Negamax and Alpha-beta only kick in when there are few holes left, one could set a global variable `gDotPos` to be the indices of all the positions with dots and then have `findMoves()` iterate through those positions instead of the entire board. This won't make significant difference for rule of thumb use of `findMoves()`, but when recursing and near the end of the game with high usage from Negamax or Alpha-beta, it could.

Othello Lab 6:

Update your Othello 5 code as follows:

A) Create a global on the **third line** of your code with a name like `LIMIT_NM` = 11 which you will use to determine whether `negamax()` should kick in (ie. if there are fewer than `LIMIT_NM` holes).
B) Print out the elapsed time in seconds with one decimal point followed by an s. The decimal point is so that the grader, looking for the final integer in the output, doesn't accidentally mistake the time for a move. Make sure that this time prints out whether or not your script is called with arguments (ie. either tournament or a specific scenario).
C) Make sure the tournament size is set to 100.
D) Run tournaments starting from `LIMIT_NM` going from 1 (which is Othello 4) all the way up to where your Negamax bogs down. If you had Ts on the last line with the grader, your number is 10. If you got only one or two Ts, your number is 11. Be sure to give yourself ample time for the last run (could be some 10s of minutes). For each run, record the time taken (in seconds) and your score. You'll be submitting this.

Now create a new Othello 6 script derived from your Othello 5 code:

E) Insert Alpha-beta code and replace your calls to `negamax()` with calls to `alphabeta()`
F) Update `LIMIT_NM` to `LIMIT_AB`, and run it from 1 up to the point where it takes over 200 seconds to run a tournament. Record each time and score.
G) Submit your code and results of parts D and F to the AlphaBeta submission form.