Negamax and Alpha Beta
AI - Fall 2018

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 [brd.count(tkn) - brd.count(enemy)]
    nm = negamax(brd, enemy)           #   game not over; tkn passes
    return [-nm[0]] + nm[1:] + [-1]

  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 three ways to speed things up (with two more ideas):
1) Cache values for Negamax, and when finding all possible moves.
2) Short circuit and return right away when there is exactly one hole and one move.
3) 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 `negamax` call, and if it is an improvement on the prior one, then it prints out the improvement.

4) If `brd` is a string (the case for most students), `makeMove` should not do individual token replacements on the string. Instead, blast the string to a list, do the token flips at the appropriate positions, and then reconstitute with a `"".join(…)`.
5) If you've been setting globals, even indirectly, `negamax` will not play nice with your code. Recursion together with setting globals are a big no-no.

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 [brd.count(tkn) - brd.count(enemy)]
    ab = alphabeta(brd, enemy, -upperBnd, -lowerBnd) # game not over
    return [-nm[0]] + nm[1:] + [-1]

  best = [lowerBnd-1]
  for mv in possible moves for tkn:
    ab = alphabeta(makeMove(brd,tkn,mv), enemy, lowerBnd, upperBnd)
    score = -ab[0]
    if score > upperBnd: return [score]    # Vile to the caller
    if score < lowerBnd: continue          # Not an improvement
    update best
    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 Alpha Beta going down a particular branch, after converting it from the enemy's point of view to that of tkn, there are three scenarios. The middle scenario (which also holds in Negamax sans Alpha beta), is that the score is not as good as the best score from a prior branch. In other words, `tkn` has done better earlier; therefore this new branch is not of interest. The remaining scenarios are when the best prior branch can be improved upon. Usually (the last scenario, which also holds in Negamax sans Alpha beta) this means that the code updates what it intends to return, updates the lower bound (that is to say, this branch is now the best branch), and continues processing.

However (the remaining option, which only holds with Alpha beta) there is a case of too-much-of-a-good-thing. 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 can be ignored (pruned).

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, 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 AlphaBeta 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 AlphaBeta, it could.